October 1979

Using the iSBC 544™ Intelligent Communications Controller

Steve Verleye
OEM Microcomputer Systems Applications

Using the iSBC 544 Intelligent Communications Controller

Contents

I. INTRODUCTION

As the microcomputer system found its way into more and more demanding applications the need became clear for a new and innovative solution to the old problem of providing timely response to real world events. This need was never clearer than in the field of communications where throughput and response time are the keys to success. The iSBC 544 Intelligent Communications Controller (ICC) is the vanguard of a family of intelligent slave computers that provide a unique and powerful answer to the needs of the microcomputer user.

This application note is intended to introduce the reader to the intelligent slave concept in general and the iSBC 544 board in particular. After a brief overview of the evolution of the concept and the features it provides, the hardware and software aspects of the controller are studied. Following this a summary of various system throughput tests is examined to evaluate the performance of the intelligent slave versus more traditional system architectures. We then study two example applications of the product and relate the earlier discussions to the real world. Finally, some system software is presented that handles all data transfer duties between master single board computers and intelligent slaves on the MULTIBUS system bus. More detailed information on many of the topics covered in this note can be found in the related publications listed in the front-piece.

II. OVERVIEW

Intelligent Slave Architecture

Over the years, component technology has increased at a rapid pace going from discrete components (eg. transistors) to integrated circuits (eg. TTL devices) to programmable peripheral controllers (eg. Intel 8251A Universal Synchronous/Asynchronous Receiver/Transmitter) to fully intelligent slave devices (eg. Intel 8041A Universal Peripheral Interface). At the system level the evolution followed a similar path using the increasing component technology to create more and more powerful system building blocks. The iSBC 508 I/O board used TTL logic to provide digital I/O expansion for iSBC computers. The iSBC 534 board took advantage of programmable LSI devices to provide a programmable communications expansion board. Now, with the advent of the iSBC 544 Intelligent Communications Controller, a new level of system capability is made possible with the fully intelligent slave controller.

The cornerstone of the intelligent slave architecture is the dual port memory. Through the use of this shared memory space, a fast and efficient protocol can be established to allow for cooperation between master and intelligent slave in solving the needs of the application system. In addition to the shared memory, the CPU on the intelligent slave also has some local RAM and local PROM storage for programs. By using this architecture the advantages of multiprocessing and Direct Memory Access (DMA) controllers are blended together. Unlike DMA controllers, the intelligent slave works totally within its own data space. Therefore, it is not affected by bus traffic nor does it add to this traffic. And, since the on-board CPU gets its instructions from local PROM instead of predefined hard-wired logic or microcode, the user has total flexibility in defining the functions the intelligent slave will assume in the overall system.

Although the contents of an intelligent slave make it look very similar to a single board computer, the assumption of the slave role provides a distinct advantage. By performing duties for a master single board computer, the slave relieves the master of low-level processing duties and at the same time is itself relieved of system responsibilities.

In order to position the iSBC 544 product and outline what features it brings to the application system it is necessary to define the functions involved with communicating data. The three main functional divisions are illustrated in Figure 1. At the lowest level the physical interconnection is maintained. This level involves such standards as RS232C which defines the requirements for transmitting bits from point to point.

The data transmission level involves the transfer of bytes and/or blocks of data from devices to computers and from node to node in computer networks. The hardware dependent software such as interrupt service and device polling is

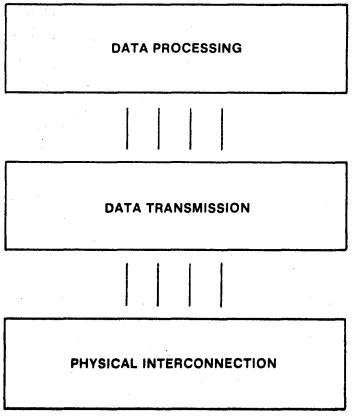


Figure 1. Layering of Communication System Functions

part of this level as are handlers for standard protocols such as SDLC, HDLC, Bisync and X.25 or special purpose schemes and custom protocols.

The highest level performs the actual processing of the data and calls upon the lower levels to move the data from place to place. The application software resides at this level as do some high level software functions such as program to program and process to process communications packages.

Now that we have a map of system functions to guide us, it is possible to gain an understanding of the usefulness of a product like the iSBC 544 Intelligent Communications Controller. If an iSBC 534 board (which contains four USART devices) was included to handle the expansion of serial I/O capacity the mapping of system functions would look like that shown in Figure 2. The four USARTs on the board would handle the physical interconnection but due to the lack of intelligence on the board the master CPU would be burdened with all of the data transmission duties in addition to its real duty, data processing.

When an iSBC 544 board is used in the system, the mapping of system functions is as shown in Figure 3. The physical interconnection is still handled by the USARTs on the board but now the on-board CPU can be programmed to assume the data transmission duties. With an intelligent slave in the system, the master CPU is freed to concentrate on the data processing functions and the end result is that each function in the system is handled in the most efficient manner possible.

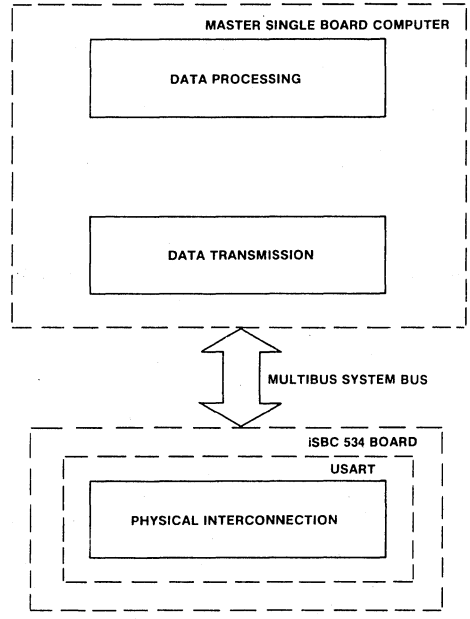


Figure 2. Mapping of System Functions with iSBC 534 Board

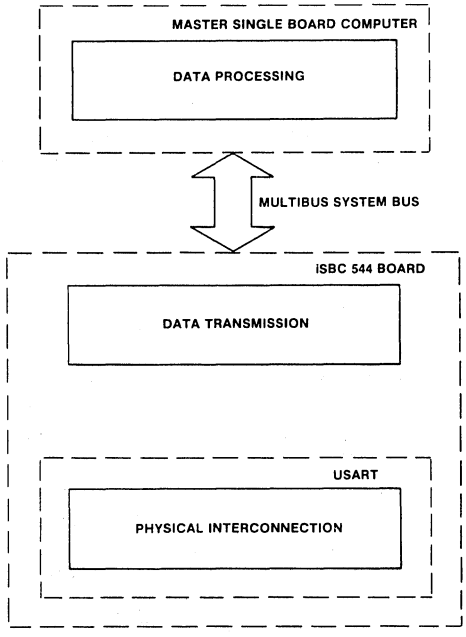


Figure 3. Mapping of System Functions with iSBC 544 Board

The iSBC 544 Board

The iSBC 544 Intelligent Communications Controller contains:

- An Intel 8085A CPU operating at 2.76 MHz.
- Sockets for up to 8K bytes of read only memory (user can choose Intel 2716, 2316E or 2732 devices).
- 16K bytes of dynamic, dual port Random Access Memory (RAM).
- 256 bytes of static local RAM.
- Four Intel 8251A USARTs with programmable baud rates.
- Two Intel 8253 Programmable Interval Timers.
- Intel 8155 parallel interface providing 22 parallel I/O lines and one 14 bit interval timer. Various input and output lines are dedicated to provide an interface to a Bell 801 or equivalent Automatic Call Unit (ACU).
- 8259A Priority Interrupt Controller.

III. HARDWARE CONSIDERATIONS

This section of the application note will focus on the iSBC 544 hardware and will outline the features of the board and its uses. Appendix A contains simplified logic diagrams of the iSBC 544 board which can be referenced in the following discussions.

Two Mode Operation

The iSBC 544 board is capable of operating in one of two modes; 1) intelligent slave and 2) stand-alone communications computer. The mode can either be set with a switch or it can be "toggled" via a software driven flip-flop on the board. In the intelligent slave mode the CPU on the iSBC 544 board operates strictly within its on-board resources. Communications with 8-bit and 16-bit master single board computers is accomplished through the dual port memory. Since the on-board CPU executes code out of its local PROM program storage the system designer is free to define which functions the slave will assume in the system design. As discussed earlier, this could include all or part of the system data transmission duties or could involve application specific duties such as terminal format control, code conversion or terminal input editing.

In the stand-alone mode, the logic on the board disables off board access to the dual port RAM and the bus buffers are used to allow the on-board CPU to access expansion memory and I/O on the MULTIBUS system bus. In this mode the iSBC 544 board drives the bus busy (BUSY/) control line active disallowing any other bus master access to the bus. The stand-alone communications computer is capable of performing all of the functions of the applications system. Referring once again to the diagram of the functions of a communication system, the stand-alone communications computer, with or without system expansion, is responsible for all data transmission and data processing functions. In small applications requiring multiple serial lines the stand-alone iSBC 544 controller is a perfect fit.

In very special circumstances it is possible to share the system bus by toggling the mode set flip-flop between master and slave mode. Figure 4

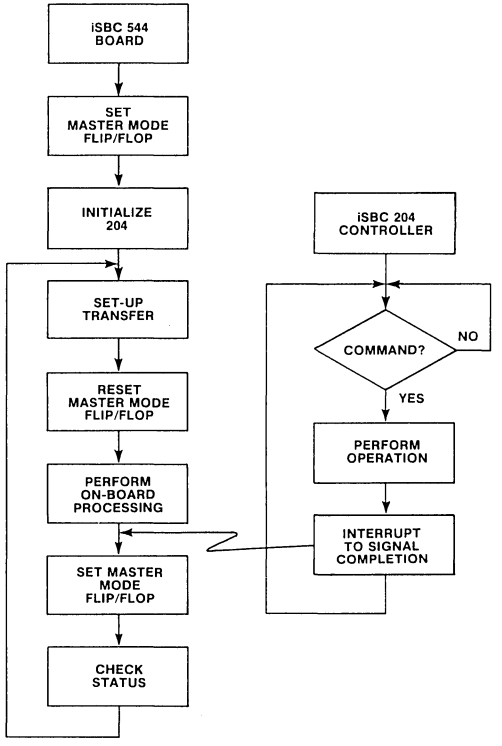


Figure 4. iSBC 544 Controller Running iSBC 204 Disk Controller

shows the flow chart for a routine (code in Appendix B) that makes use of the "software switch" to operate an iSBC 204 Diskette Controller. Using the iSBC 544 board in a system with DMA devices is not recommended except in cases where DMA accesses are short and relatively rare. The use of the CPU for the handling of other system devices could seriously degrade its performance as a communications controller. However, this capability could be extremely useful in a system such as a small message store and forward where the disk traffic is not heavy and including a CPU card just to handle the disk would be wasteful. Use of the "software switch" to share the bus with another iSBC CPU is not advised because of the amount of protocol that would be required to keep the CPUs from interfering with each other on the bus.

Dual Port RAM

Figure 5 illustrates the dual port RAM memory array on the iSBC 544 card. A triple bus architecture is used to allow other MULTIBUS bus masters access to the RAM on the intelligent slave. Both the on-board CPU's bus and the MULTIBUS system bus are connected to the dual port controller. From here the dual port bus is connected to the 16K of dynamic RAM memory. Memory transfer requests from either of the first two busses are handled by the dual port control logic with the on-board CPU being given priority if contention arises. The local CPU is favored so that it is not overly delayed in handling its time critical functions.

The address mapping of the dual port memory on the iSBC 544 is diagrammed in Figure 6. The user can enable access from the MULTIBUS system bus to 0, 4K, 8K or all 16K of the RAM on each iSBC 544 board. The dual port control logic decodes the full 20-bit address and provides an 8-bit data path to the bus. For these reasons the iSBC 544 board is compatible with 8080A, 8085A and 8086 based single board computers. The user can also select the block of addresses on the system bus to which the iSBC 544 RAM will respond.

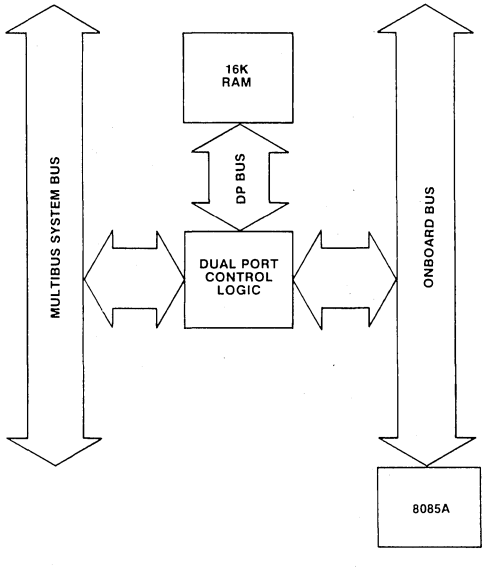


Figure 5. Dual Port Control Logic

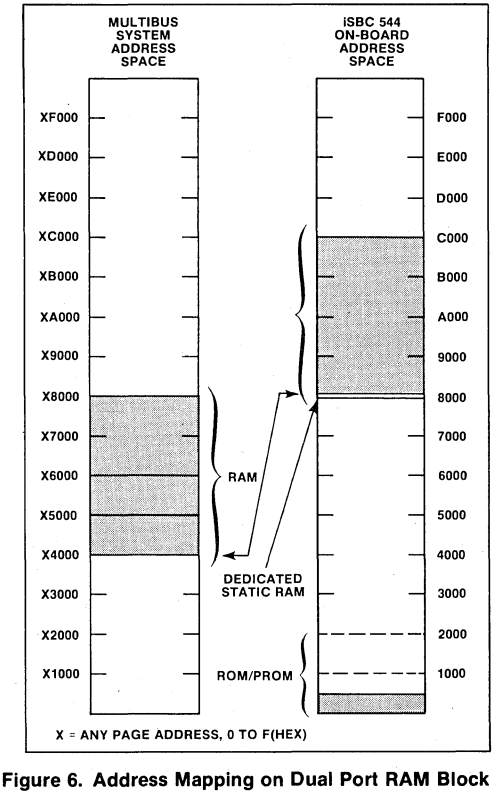


Figure 6. Address Mapping on Dual Port RAM Block

When accessed by the on-board CPU, the dual port RAM always appears at 8000H. If the iSBC 544 board is operating in the stand-alone computer mode, the board is capable of generating the 16-bit bus address supported by the 8085A CPU.

Interrupt Structure

The interrupt structure of the iSBC 544 controller is designed to handle the heavy load imposed by the inherent real-time nature of the communications application. An 8259A Priority Interrupt Controller handles the four receiver and transmitter ready interrupts from the 8251A devices and provides vectored interrupts using one of many available priority schemes. In addition to the eight interrupt sources handled by the 8259 there are various others that can be connected directly to the vectored interrupt inputs on the 8085A (RST 5.5, 6.5, 7.5 and TRAP). One interrupt is generated by the dual port control logic whenever a byte is written into the base address of the dual port memory by an offboard CPU. This interrupt, the flag interrupt, is cleared automatically when the on-board CPU reads the byte and is useful when designing a master-slave protocol since it provides a unique interrupt to each slave in the system.

If the 8251A devices are used to interface to modems the loss of carrier and ring indicator interrupts from all four channels need to be connected to 8085A interrupt request inputs. This is accomplished with four input OR gates tying the eight sources into RST 6.5. The ring indicator and carrier detect lines can also be monitored through a parallel I/O port. This port would be read in a polled system to determine status or could be used along with the OR-tied interrupts to determine which channel is sourcing the current interrupt.

The remaining interrupt sources come from the extra timer/counters and from the MULTIBUS interrupt lines. In addition to receiving interrupts from the bus, the iSBC 544 board has the capability of generating MULTIBUS interrupts using the Serial Output Data (SOD) line on the 8085A CPU.

Modem and Autocall Interface

The iSBC 544 controller uses 8251A and 8155 devices for interface to modems and an autocall unit respectively. All of the necessary handshaking signals concerned with the modem interface are connected to the 8251A and the carrier detect and ring indicator signals, as previously mentioned, can be connected to interrupt inputs. The 8155 parallel ports are wired as shown in Figure 7. All of the commonly used signals defined in the EIA RS-366 specification for interface to an autocall unit are provided. The software necessary for handling the ACU becomes a simple matter of responding to the ACU requests and sending out the BCD digits representing the number being dialed. In addition to the ACU interface, the 8155 monitors various signal states and provides software reset capabilities for the USARTs and some interrupts.

IV. SOFTWARE CONSIDERATIONS

Software for the iSBC 544 ICC falls into three main categories; device programming, master-slave protocols, and communications support. Each of these three topics is covered in the following section with the aim of defining the software requirements and functions of the iSBC 544 board.

Device Programming

The main sources of the power and flexibility of this product are the programmable LSI devices on the board. The first duty of the on-board software is programming these devices to handle the specific task at hand. To start with, the 8251A USART can be programmed for synchronous or asynchronous operation. In synchronous mode the user specifies even, odd or no parity and either external or internal sync detect with one or two sync characters. In the asynchronous mode the programmer selects the parity, the character length (5, 6, 7 or 8 data bits), the framing control (1, 1½ or 2 stop bits) and the baud rate scaling factor (input clock frequency divided by 1, 16 or 64).

The 8253 Programmable Interval Timers provide the receiver and transmitter clocks for the USARTs and, along with the 8251A baud rate scaling factor, are programmed by the software to provide the desired communications frequency. In addition, two additional 16 bit timers are left available to the applications programs to be used as event counters, real-time interrupts, etc.

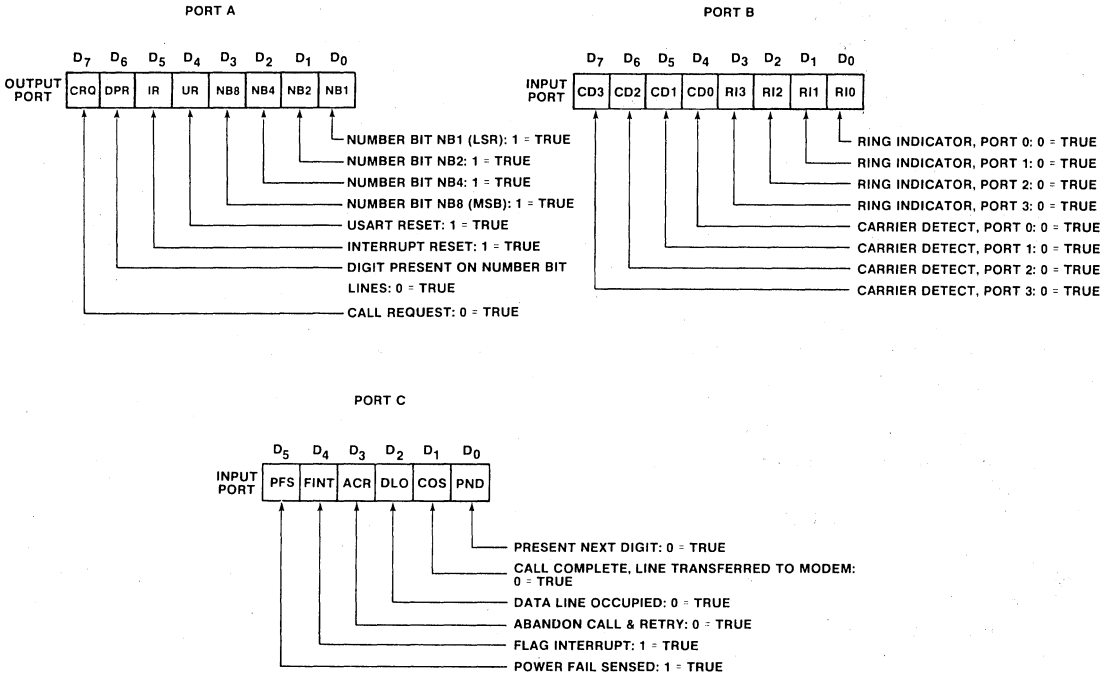


Figure 7. 8155 Pinout Definitions

The 8259A Priority Interrupt Controller is programmed to vector all interrupts through a jump table in memory. Also, the device provides software selectable priority schemes and an interrupt mask register for sophisticated interrupt management designs.

Last, but not least, the 8155 Programmable Peripheral Interface provides various software controlled input and output ports as discussed in previous sections. One specific point to remember is that the power on state of the 8155 clamps the reset signal to the USARTs active and must be removed by programming the 8155 before communications can begin.

Master-Slave Protocols

If an application system is visualized at the highest level it appears to be a computer with various inputs and outputs as depicted in Figure 8a. If this computer is broken down into a master CPU and one or more intelligent slaves, great increases in efficiency and system throughput can be realized by distributing the duties between the CPUs (Figure 8b). Once this split is performed, some well defined means of communication between master and slaves needs to be defined so that the processes that execute on the different machines can cooperate. This means of communication takes the form of a protocol followed by both master and slave.

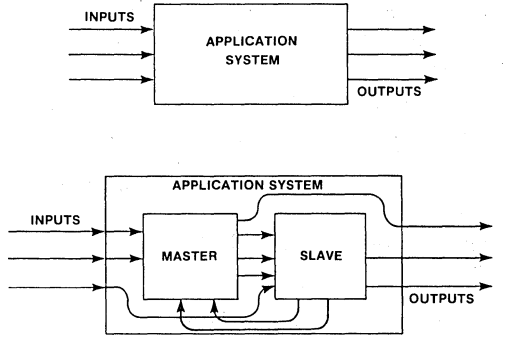


Figure 8a and 8b. System Software Block Diagrams

The intelligent slave architecture was designed to simplify the development of the necessary protocol. The shared memory space in the dual port RAM provides a large communications buffer area where data and commands can be transferred using normal memory transfers. Data structures of any needed complexity can be built in this memory area and accessed by both master and slave. The flag interrupt can be used to provide a unique synchronization signal from a master to a given slave. In addition, the MULTIBUS interrupt lines can be used to provide extra signals in both directions. As we shall see in the system software section, these basic tools can be utilized to design a general purpose data transfer mechanism which isolates the applications processes from the worries of protocols and synchronization.

Communications Support

The previous software topics dealt mainly with the system overhead that must be handled by the communications processor. The larger and more important duty of the CPU is dealing with the application at hand—communications.

When configured as an intelligent slave to some master iSBC CPU board, the iSBC 544 board works to offload the master of communications related functions and at the same time is itself relieved of a major share of the system overhead and can be tuned to provide the highest possible throughput. With this combination, more complex applications can be tackled where the number of lines and the line frequencies are greatly increased. Multiple systems can be employed to provide a network facility with the iSBC 544 board now handling the network protocol in addition to its other duties. The architecture of the iSBC 544 controller is designed to simplify the user's software development process. The board can be programmed to handle many possible data transmission functions from simple line protocols to terminal control to link protocols and all the way up to network protocols.

In the stand-alone mode, the iSBC 544 board can assume total responsibility for the application. This can be done with on-board resources only or can include the support of offboard expansion like the iSBC 534 four channel serial controller. Applications of the stand-alone controller could include cluster controllers, peripherals managers, line concentrators or any other small system.

V. THROUGHPUT ANALYSIS

This section of the application note deals with studies that have been done to quantify the performance of the iSBC 544 board in both the stand-alone and intelligent slave modes. After describing the various test configurations and assumptions the data will be presented in graphical form and analyzed. The graphical data can be found in Appendix C.

Stand Alone Throughput

The first two tests were run to determine the absolute best case throughput of the iSBC 544 board configured as a stand-alone computer. Figure 9a shows the iSBC 544 controller continuously outputting data from four buffers to the four USARTs. Figure 9b shows essentially the same setup with eight channels, four on the iSBC 544 board and four on the iSBC 534 expansion card. In each configuration the 8251A was run in synchronous mode and the baud rate was incremented until the transmitter empty signal from the USARTs became active. Further increments of the baud rates would not have resulted in higher throughput since the CPU was already spending 100% of its available time responding to USART service requests.

The maximum rate for the first configuration (iSBC 544 board only) was 32,311 baud per channel. When the iSBC 534 expansion board was added a rate of 12,186 baud per channel was achieved. The drop in baud rate was due to the extra processing required by the offboard logic (eg. reading 8259 interrupt controller on the iSBC 534 board to determine which device is requesting service).

It should be noted that the serial throughput tests were run with almost no overhead and no actual processing of the data involved. The reader is expected to apply information on the amount of overhead expected in each individual application. For instance, if the application code for a given system is expected to utilize approximately 40% of the available CPU time and we wish to run four

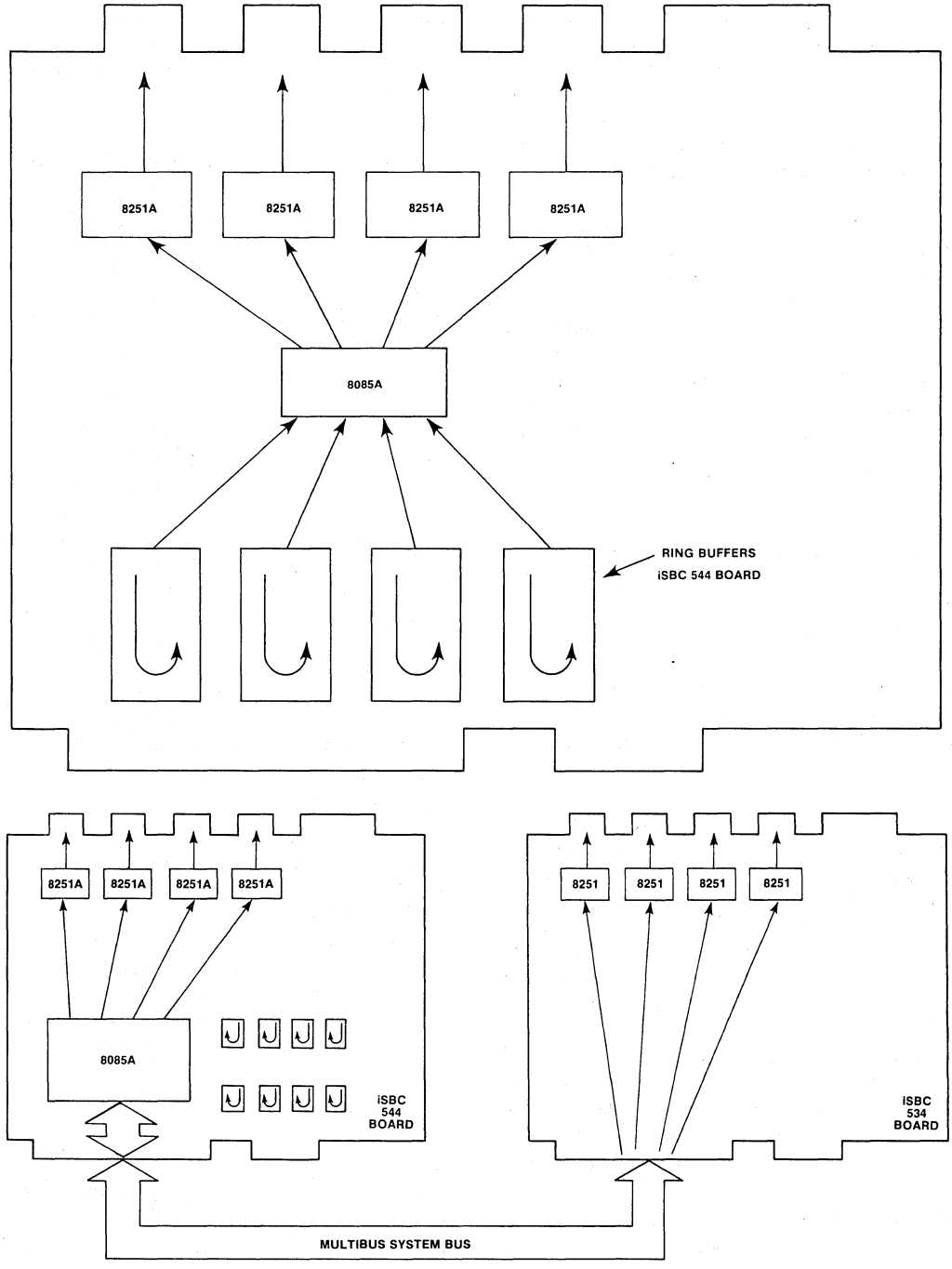


Figure 9a and 9b. Stand-Alone Throughput Configurations

full duplex channels in asynchronous mode the estimate of maximum baud rate would take the following form.

32,331 baud per channel — 40% = 19,398.6 baud

19,398.6 baud per channel synchronous x 10/8 = 24,248.25 baud asynchronous

24,248.25 baud per channel half duplex/2 = 12,124.125 full duplex

Therefore, the maximum standard baud rate would be 9600 baud per channel in full duplex asynchronous mode.

Intelligent Slave Throughput

The remaining four configurations were set up to determine the effectiveness of the intelligent slave in the overall system. The general system configuration is illustrated in Figure 10. The boards surrounded by the box represent the systems under test. The disk controller and two iSBC 80/20 single board computers were active on the bus to simulate the normal bus traffic load in an application system. Various bus duty cycles were created using the computers and the disk controller to perform tasks that resulted in fixed bus utilization.

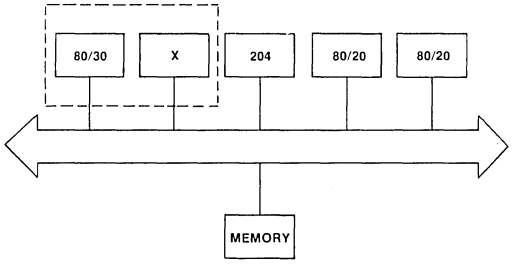


Figure 10. General System Configuration for Throughput Testing

In each configuration a single full duplex channel was set up with the input provided by another CPU. Only those functions dealing with system overhead were included and the data measured reflected the amount of bus time, master CPU time and slave CPU time left available to applications oriented tasks. In each case this percentage of time available was measured as the baud rate was stepped up so that a graph could be constructed showing time available as a function of transmission speed.

CPU free time was measured using a counting program running in the background. After each USART interrupt the counter was started. As interrupts from other sources came in the counting was preempted and then resumed after servicing the interrupt. When the next USART interrupt occured, the counter contents were examined and if the value was lower than the stored value the current value became the stored value. After ten minutes the stored value was retrieved and used as an indicator of the worst case time available between interrupts.

System bus utilization was measured using the circuit shown in Figure 11. The voltage measured by the digital voltmeter represented a time average of the voltage at the output of the flip-flop. A calibration chart was created using a pulse generator to simulate various duty cycles and then this chart was used to measure bus activity while the test was running.

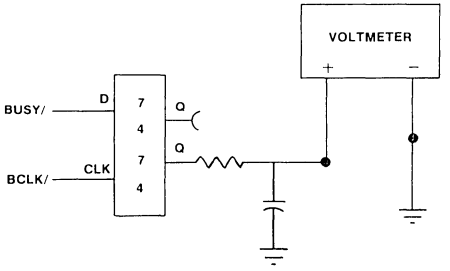


Figure 11. Bus Free Time Measurement Circuit

Configuration 1 is shown in Figure 12. This system uses a typical method of communications expansion with the iSBC 80/30 single board computer handling the lines directly via the serial I/O ports on the iSBC 534 I/O controller board.

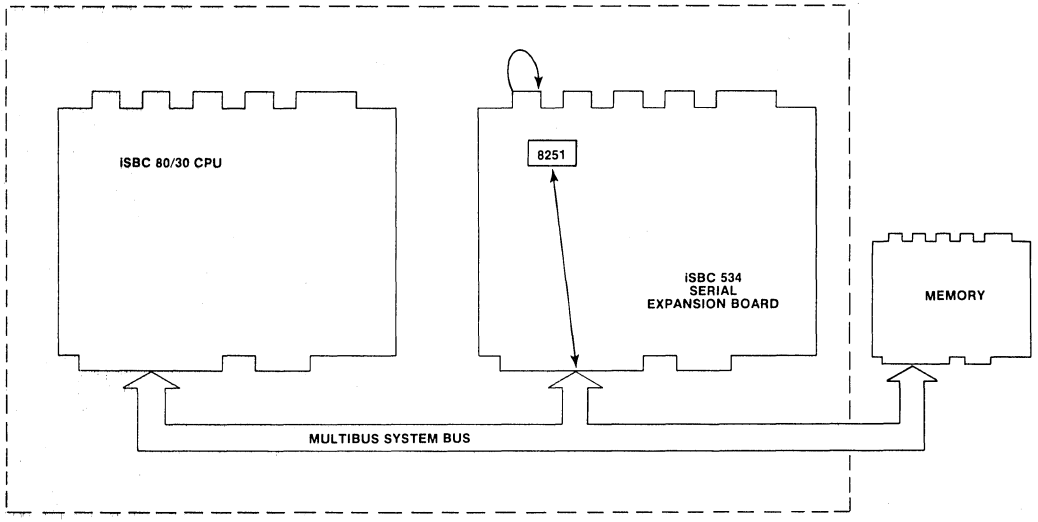


Figure 12. System Throughput Test, Configuration 1

The second configuration (Figure 13) illustrates the performance of the traditional DMA controller approach. If the communications controller had DMA logic instead of a dual port memory and transferred data directly into system memory the performance would be as observed in this test.

In configuration 3 (Figure 14) the iSBC 544 board was used in the intelligent slave mode. This configuration differs from the second in that memory transfers involved only local memory and bus access was not required on a per character basis.

The fourth and final configuration sought to identify the loading that additional intelligent slave controllers would impose on master CPU time and bus free time. Figure 15 shows the

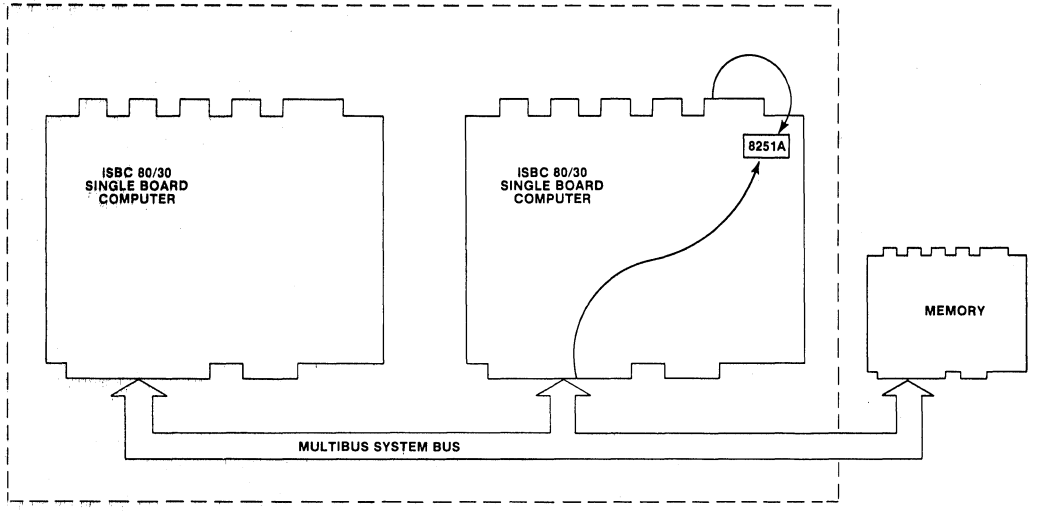


Figure 13. System Throughput Test, Configuration 2

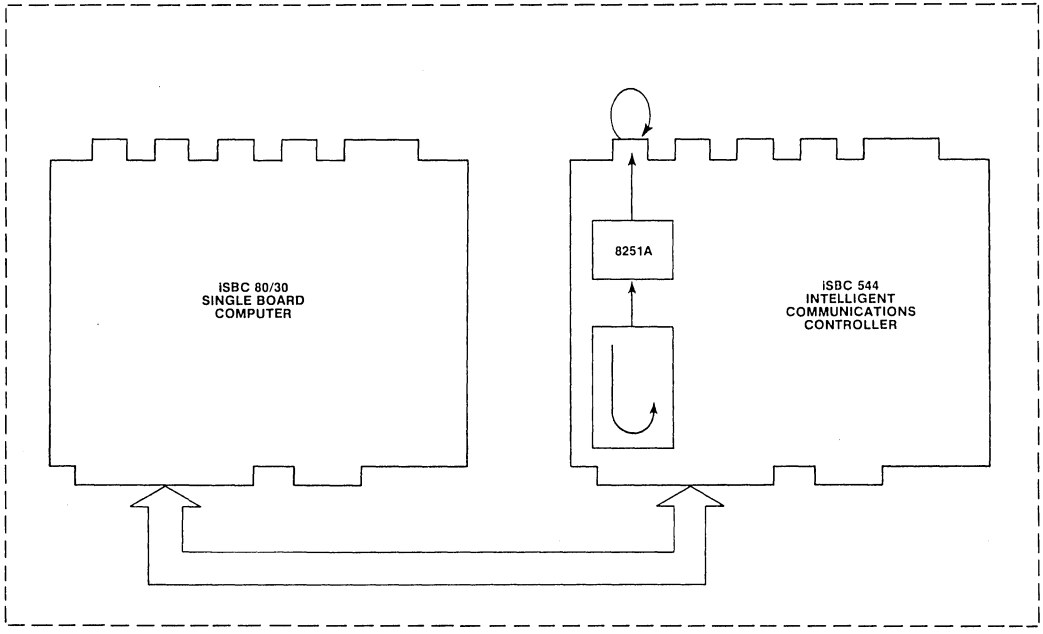


Figure 14. System Throughput Test, Configuration 3

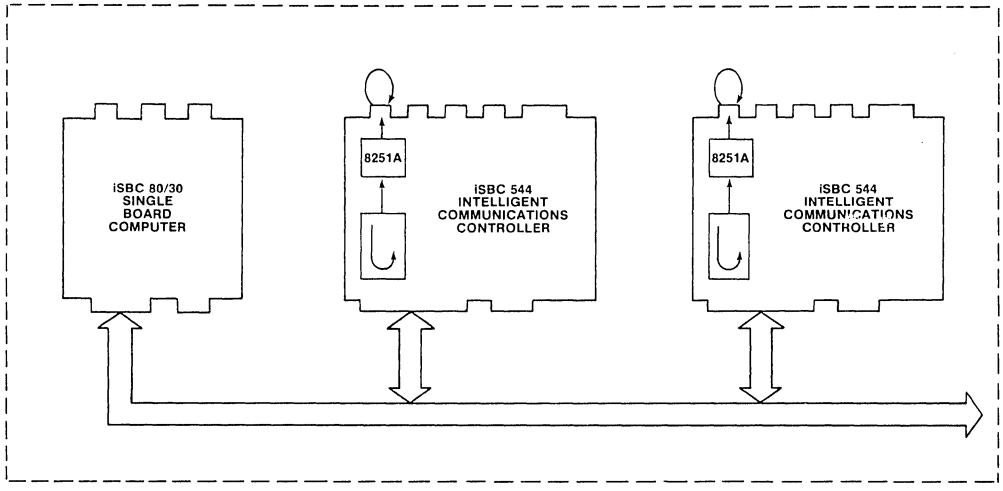


Figure 15. System Throughput Test, Configuration 4

configuration with two iSBC 544 boards executing identical programs.

The graphical presentation of the results is split into two sections. The first three graphs (Graph 1 through Graph 3) show the relationship between baud rates and the master CPU, system bus, and slave CPU utilization. All of these results are based upon tests with 30% induced bus traffic (i.e., the two iSBC 80/20 computers and the iSBC 204 disk controller were active.)

In graph 4, processor free time is graphed as a function of bus traffic. The processor in this case is the one actually involved with the data on a per character basis (i.e., iSBC 80/30 board in configuration 1, iSBC 80/30 board simulating DMA Controller in configuration 2, and iSBC 544 board in configuration 3).

Finally, graph 5 illustrates the maximum attainable baud rate for each configuration as the bus traffic is increased.

All of the graphs identify the relative performance difference between the configurations. Absolute numbers are not presented due to the fact that the overhead imposed by the test software affects the CPU time being measured. Since the overhead applies equally to all configurations, the relative performance indications are valid.

Based upon the data presented, the DMA controller and intelligent slave use 3 times less CPU time than an I/O controller. Also, the iSBC 544 intelligent slave generates 12% and 6% less bus traffic than the I/O controller and DMA controller respectively. Finally, the intelligent slave uses 8% less slave CPU time than the DMA controller approach.

The earlier discussion that dealt with the intelligent slave architecture pointed out that the distribution of intelligence would offload the master CPU so that it would retain sufficient processing power for the actual application, whatever that may be. In addition, it was stated that the assumption of the slave role would relieve the slave CPU of system overhead and at the same time reduce system bus traffic. All of these assumptions are supported by the results of the testing presented here.

The second set of graphs identify the effects of bus traffic on the performance of the various components of the system. The main observation to be made in this sequence is the drop in CPU free times and maximum baud rates that occurs when the bus gets busy. This effect is observable in the communications processor free time when the iSBC 534 expansion board or the DMA controller configuration is used. No effect is evident in the configuration with an iSBC 544 board.

The cause of this effect is the amount of bus access required by each configuration to move the characters from the USART to or from the buffer. With an iSBC 534 board the master CPU receives an interrupt, polls the offboard 8259 interrupt controller, reads in a character, stores it in system memory and sends an end of interrupt command to the offboard interrupt controller. When the iSBC 80/30 computer receives an interrupt all processing is performed onboard until a bus access is required to move the data byte from/to memory. In the case of the intelligent slave, all processing for a character is performed onboard. Thus, as the system bus becomes very fully utilized, the delays encountered in receiving bus access by the first two configurations become significant.

The fourth configuration, which was set up to test the effects of adding more intelligent slaves, shows that extra slaves cause no appreciable increase in system load. All of the data points for two slaves were identical to the points for one slave in graphs 1 through 5.

VI. APPLICATION EXAMPLES

A Distributed Control System

The potential applications for a product like the iSBC 544 communications controller are almost unlimited and not restricted to the traditional Data Communications market. The first application example that is studied concerns industrial automation. Due to the fact that the system is distributed and requires a generalized network, the iSBC 544 board is a natural prospect to handle the communication links between the various nodes in the system.

Design Requirements

The system to be designed is intended to provide the framework for a family of distributed control systems where the configurations and the objects to be controlled vary from system to system. Figure 16 shows the general picture of the system.

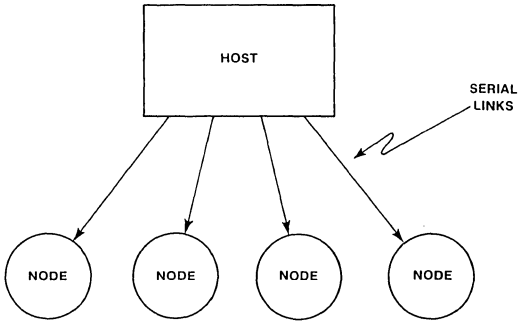


Figure 16. General Diagram of Distributed Control System

The host is responsible for providing supervisory control and a high-level human interface. The system can be expanded as shown in Figure 17 where the controllers attached to the host are replaced by intermediate nodes which contain controllers or other nodes. This process can be continued as far as is necessary to provide the needed number of controllers. Each controller in the diagram represents a localized closed loop control system that is tailored to the specific application.

The following system requirements need to be met by the computer network:

- The host CPU must have sufficient computational power to handle the human interface, mass storage management, supervisory control calculations and network control.
- The host CPU must not be overly burdened by low-level communications functions if it is to handle the other duties assigned to it.
- Node controllers must be capable of handling 8 medium speed lines and also modems and autocall units since the nodes or controllers attached may be remote.

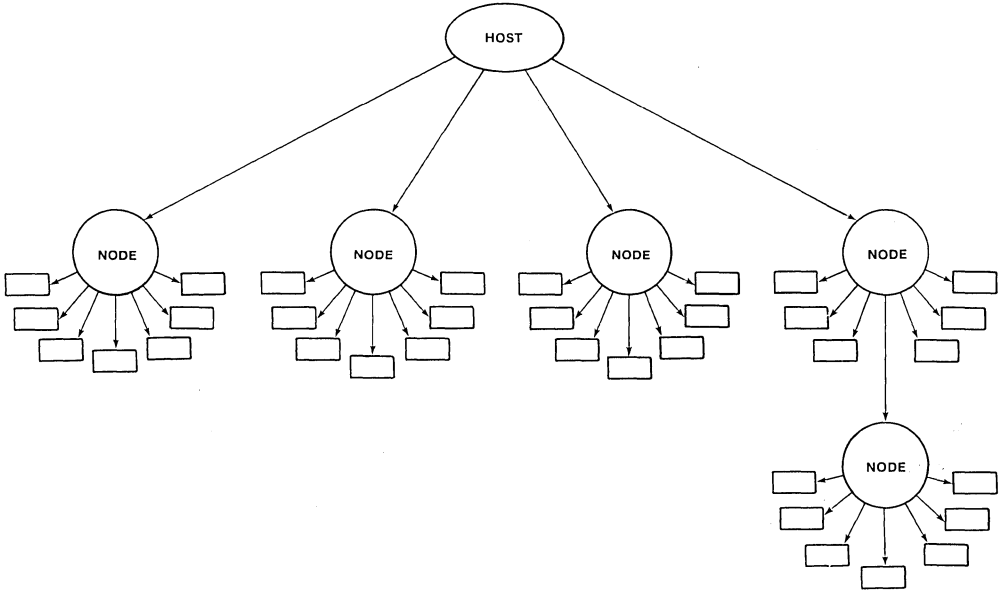


Figure 17. Expanded Diagram of Distributed Control System

- The message transmission format must be independent of the configuration and end application. The nodes in the network must be capable of passing through messages with and without interpreting the contained data.
- The system must be capable of auto-configuration (since the network configuration is tailored to the specific application, the host must be able to automatically determine the setup at power on).
- Each node controller is responsible for verifying the integrity of the nodes attached.

System Configuration

Based upon the design criteria and the benchmark information the chosen configuration uses an iSBC 86/12 Single Board Computer as the host with an iSBC 544 intelligent slave handling the communications load for the CPU. The USART on the CPU board will talk to the local terminal and an iSBC 206 Hard Disk Controller will be used to provide up to 40 Megabytes of mass storage capacity.

The requirements for the node controllers point to an iSBC 544 board configured as a stand-alone communications computer with an iSBC 534 board as expansion to provide the necessary 8 lines. The throughput data indicated a raw throughput value of 12K baud on each channel. With the data rates expected being far below this, sufficient time will be left over for background functions. Thus, the software requirements for each node can all be met by the CPU on the iSBC 544 board and the inclusion of an expansion board does not necessitate another iSBC computer.

A typical controller in the system would look like that shown in Figure 18. The iSBC CPU handles the local closed loop control, using parametric information sent from the host. This information would typically include setpoints, tolerances and alarm limits. The serial channel on the CPU will be used to maintain the link to the next level in the network.

Preliminary Design

The message format that the system uses is shown in Figure 19. When multiple nested levels

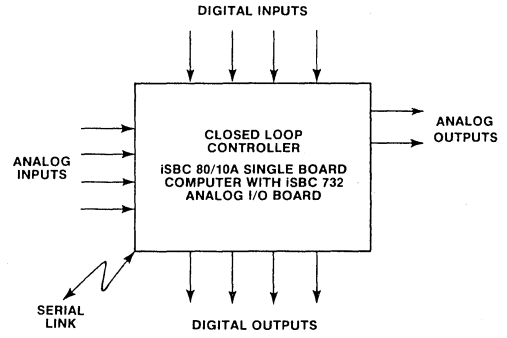


Figure 18. Typical Controller in Distributed System

Figure 19. Message Format

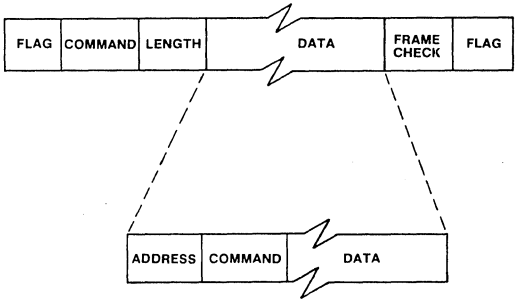


Figure 20. Nested Level Address Information

of nodes are used the data area of the message contains command and address information for the next level down (Figure 20). Interpretation of the commands in a given message is done on an individual basis except for a set of system-wide commands (eg. IDENTIFY is a system command meaning respond with your ID code). The flexibility afforded by this scheme can be extremely useful in a system where the end applications and configurations may be quite diverse (eg. a node controller that is processing a transmit command may be the only one that knows that it is sending to another node via a phone line and thus it interprets the contained data differently than another node would). The level of intelligence and the ease of programming of the iSBC 544 board make this generalized transmission scheme possible.

The simplest means of auto-configuration requires each controller in the system to send an identity message to the nearest node. This node would know the logical address of the controller that sent the message and would attach this address to the message and retransmit it to the next level as illustrated in Figure 21. This process would be repeated until the host is reached and would contain, at this point, all necessary address information to reach the given controller.

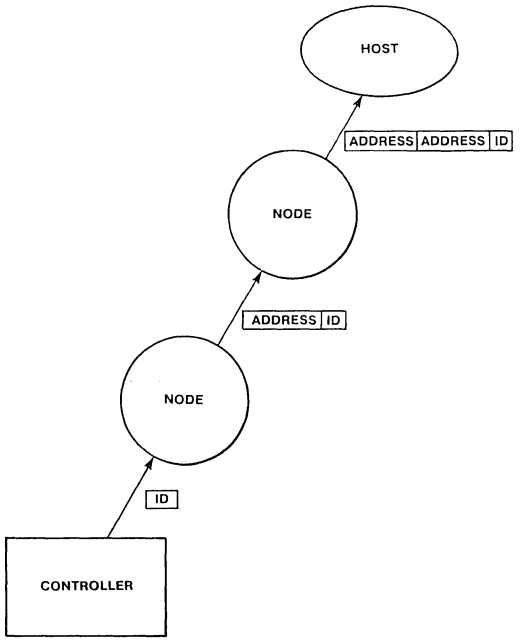


Figure 21. Auto Configuration

The human interface on the host would provide a mapping mechanism to attach meaningful symbolic names to the various nodes in the system. This labeling, along with the application specific control algorithms, make it possible to say something like "lower the temperature on the third floor to 68°F". The host breaks this information down into setpoints and tolerances, uses the map to determine the path to the node(s) responsible for the third floor and transmits the information through the network.

Each node controller in the system has the added responsibility of verifying the integrity of all the nodes attached to it. This duty can be handled by periodic background commands issued from the host and propagated through the network. Each node is responsible for passing the command along and also polling the nodes attached to it and reporting back any error conditions.

Summary

Through the use of a powerful 16-bit iSBC Single Board Computer, various low-cost 8-bit iSBC CPUs and the iSBC 544 communications controller, a flexible and extensible distributed control system is easy to design. The dual nature of the iSBC 544 board provides both an intelligent front end to the host computer and a high-speed stand-along nodal concentrator. The ability to individually customize the software on each controller provides for an easily expandable system design.

Terminal Cluster Controller

The second application example concerns itself with a terminal cluster controller. The system shown in Figure 22 uses a number of "dumb" terminals and makes them appear "intelligent" via a local microcomputer system. The local microcomputer interfaces with the operator and accesses a local data base to provide an inquiry and data entry service. When necessary, the local microcomputer is capable of calling the host via an autocall unit and exchanging information and updates to the data base.

Design Criteria

The terminal cluster controller must meet the following criteria:

- Support must be provided for from four to sixteen operator terminals all running at rates up to 2400 baud.
- Line editing on input must be provided (delete characters, delete lines and pause output).

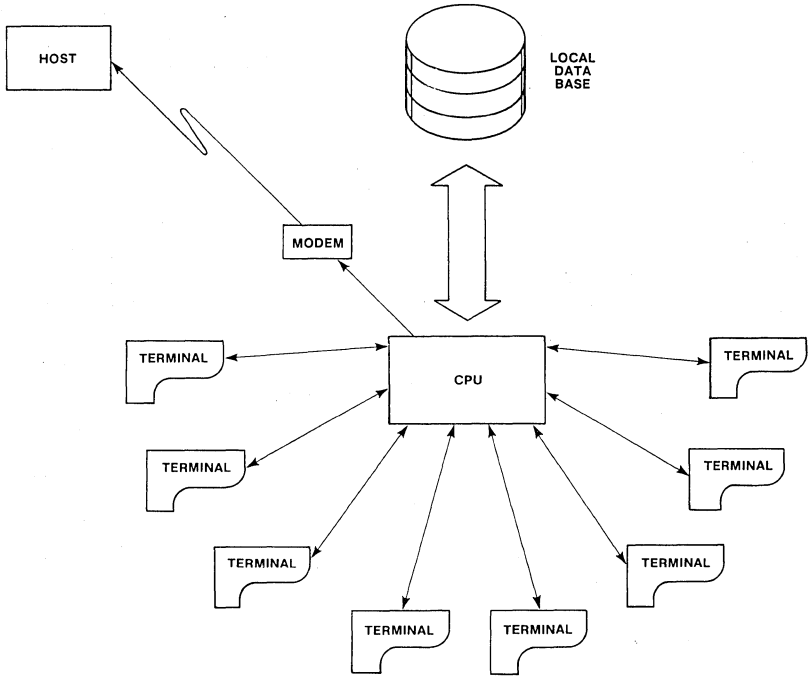


Figure 22. Terminal Cluster

- Support for the terminals must be configurable in that certain stations may require different screen formats.
- Support for an optional hard copy device must be allowed for.
- A considerable amount of CPU free time must be available after the basic terminal facilities are included. This is due to the fact that the data base management software to be written to run on the master single board computer will be extensive.
- Type ahead would be a desired feature since the processing on the master CPU after a line of input has been transmitted may cause a delay in responding and we would like to have the ability to continue entering input while waiting for the response.

System Configuration

The specific iSBC products needed to implement the system described are the iSBC 80/30 Single Board Computer with an iSBC 032 RAM Expansion Board, an iSBC 206 Hard Disk Controller and one to four iSBC 544 Intelligent Communications Expansion boards. Intel's RMX/80 Real-Time Multitasking Executive will provide the basis for the software system and will include disk file support for the iSBC 206 controller through DFS/80. The full system configuration is illustrated in Figure 23.

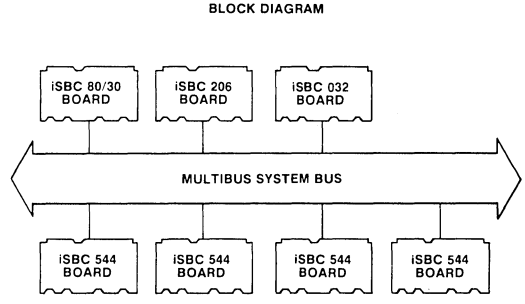


Figure 23. Terminal Cluster Controller System Configuration

Preliminary Design

The first design decision to be made involves the distribution of system functions. Due to the requirements for line-editing and type-ahead the software for processing characters input from the terminal keyboards will be somewhat lengthy. The standard terminal output handler will be very small but provisions for special screen format controls and/or hard copy devices must be allowed for. All of these requirements lead to the use of the iSBC 544 controller for all terminal functions. If the master CPU were burdened with all of these duties it would be unable to adequately perform its data base management functions. The fast CPU and 8K PROM capacity of the iSBC 544 board will be more than adequate for the task at hand.

The throughput tests indicate that the loading imposed by expanding the number of terminals (and therefore the number of iSBC 544 boards) will not adversely affect the performance of the rest of the system. Master CPU free time and bus traffic data for two intelligent slaves in the system were identical to the numbers for one slave. Thus, since the iSBC 80/30 single board computer and the MULTIBUS system bus can handle one iSBC 544 controller they can also handle the maximum of four controllers that may be required by this application. The only observable effect will be caused by the load the extra operators impose on the data base software itself.

The software needed for the iSBC 544 board is now defined and divided into three major pieces; a terminal input handler, a terminal output handler and system software to support the handlers. Since the input and output handlers are invoked via USART interrupts, all that need be done is to write a single routine for each handler and have it talk to all of the devices on the board. This can be accomplished by vectoring the proper interrupts to the entry point of the routine and then polling the 8259A interrupt controller to determine which device needs servicing.

The standard terminal input handler needs to read in the available character from the USART, check it to see if it is a special command character and, if not, store it into a buffer. If a command character is encountered, the handler will respond by performing the appropriate operation.

The standard terminal output handler simply takes characters out of a buffer upon interrupt from the transmitter and sends them to the appropriate USART. If a different output handler needs to be substituted for a special terminal or a hard copy device, a new routine can be included by modifying the interrupt vector address in the 8259A jump table.

Since the RMX/80 Real-Time Multitasking Executive is being utilized on the master CPU it is desirable to create an RMX/80 handler for the iSBC 544 boards that accepts and processes normal terminal handler request messages. In this manner, application tasks that formerly communicated with the on-board USART via the RMX/80 Terminal Handler can be made to talk to one of the devices on the iSBC 544 board by simply changing the address of an exchange. The following paragraphs, as well as paragraphs in the section on system software, assume a knowledge of the RMX/80 Real-Time Executive. This knowledge is not necessary to use the information contained in this application note. Interested readers are referred to the RMX/80 references listed in the front-piece.

Since this application can have from one to four iSBC 544 boards the RMX/80 driver will need to be configurable. A set of tasks and exchanges will be created for each terminal in the system. One task and exchange pair will accept and process terminal input request messages while another pair will process terminal output requests.

The remaining piece of software that is needed by this system will provide the means for getting commands and data between the master and intelligent slave. Since this is a common need in any system utilizing an intelligent slave we will develop a general purpose scheme that can be used by any application. In this manner, a routine such as the terminal input handler can be written without any concern for how it will get the data it is inputting to the master CPU; all it need do is call upon a standard routine to "transmit"

the data. With these thoughts in mind, the following section discusses the system software developed for master-intelligent slave communication. After the discussion of the system software we will revisit the software for the second application as an example of the use of the data transfer routines.

VII. SYSTEM SOFTWARE

In the earlier discussion of master-slave protocols, the notion was presented of developing a general purpose data transfer scheme which would enable the applications routines on both the slave and master to operate without concerning themselves with protocols and synchronization. This scheme can be implemented by designing a set of primitive routines to handle the data transfer activities. Thus, Figure 8b is expanded as shown in Figure 24 and the applications processes now call upon the primitives to handle the communications between the master and the slave.

Data Transfer Primitives

The basic mechanism used by this implementation of the primitives is a wraparound queue as shown in Figure 25. Each 8251A device has associated with it, in dual port memory, an input and an output queue each of which have a *give* and a *take* pointer. The *give* pointer contains the address of the next location in the queue that is available for filling with data. The *take* pointer contains the address of the next byte in an output queue that has been filled and is available. A queue is empty when the *give* and *take* pointers are equal and it is full when the act of incrementing the *give* pointer would make it equal to the *take* pointer. A *wrap* function is defined to increment a pointer such that an increment past the bottom of the queue "wraps" the pointer around to the top of the queue.

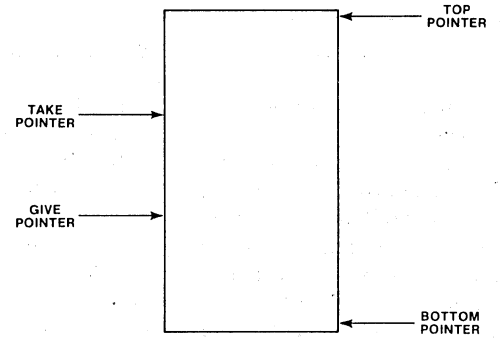


Figure 25. Wrap-around Queue Used by Data Transfer Primitives

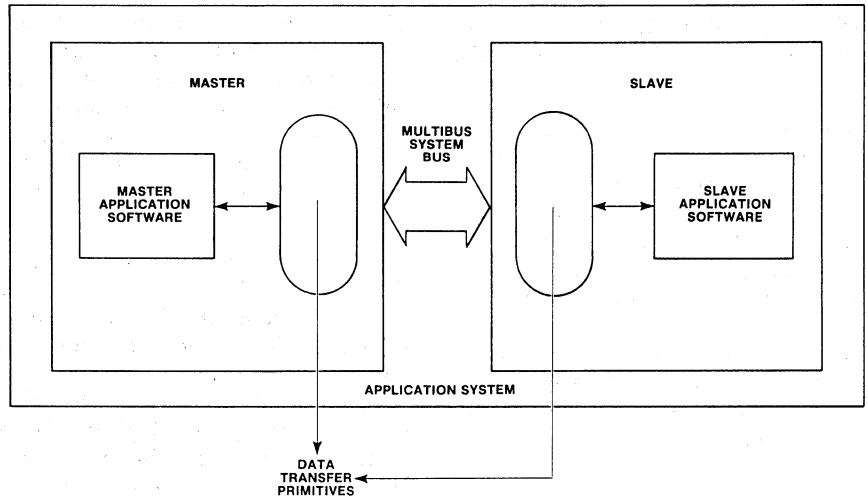


Figure 24. System Software Diagram with Data Transfer Primitives

The primitives all make use of a queue information block located at the base address of the slave's dual port memory (Figure 26). All pointer information is base relative to accommodate the needs of the two CPUs who have different memory maps. The two flag bytes carry information for master-slave and slave-master synchronization signals.

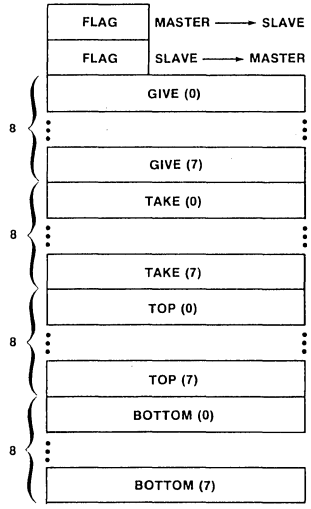


Figure 26. Queue Information Block

The set of primitives provides two distinct methods of information transfer, line oriented and byte oriented. The line oriented primitives are listed in Table 1. Both *get$line* and *send$line* transfer information between the queues and buffers provided by the caller. The disadvantage of this scheme is the number of memory moves needed to transfer information. The advantages of the line oriented method are the relative efficiencies and the simplicity of the interface from the calling routine.

The byte oriented primitives (Table 2) allow the calling routine to transfer data directly into and out of the queues. An example of the sequence for putting a character into a queue is illustrated in Figure 27. The routine servicing the receiver ready interrupt calls *next$space* to get a pointer to the next available slot in the queue and then uses this pointer to transfer the data byte directly into the queue. The *new$line, xmit, open$line* and *receive* primitives are necessary since the global *give* and *take* pointers cannot be modified until all manipulations on the affected section of the queue are complete. If the pointers were modified continuously the routine gathering the data from the other side may see invalid data.

```
/* OPERATOR TYPES "L" */

PTR = NEXT$SPACE (QUEUE$NUMBER);
VALUE = INPUT (USART$DATA$PORT);
```

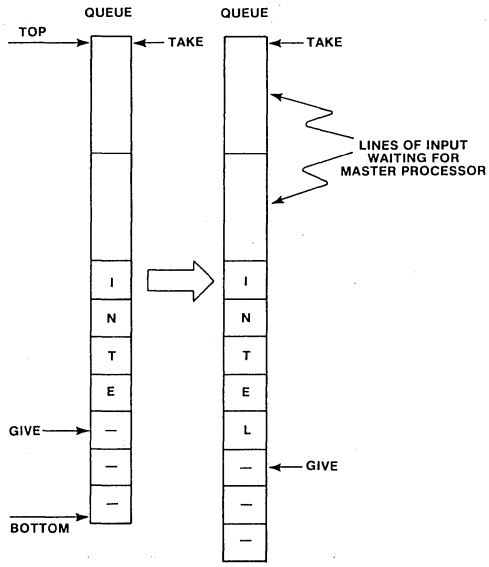


Figure 27. Sequence for Putting Data Into Queue

Table 1
Line Oriented Primitives

Primitive	Arguments	Usage
send$line	Queue$token, buf$ptr, *count* Returns: overflow	Inserts *count* characters into queue from buffer If insufficient room available, overflow indicates how many would not fit
get$line	Queue$token, buf$ptr, count Returns: Actual	Retrieves count characters from queue and puts them in buffer Actual indicates how many were actually moved

The remaining primitive routines deal with the general purpose needs of the application software with regard to interrupts, initialization and status checking. A full list of these support routines is contained in Table 3.

There are many features of this implementation and a few of them should be pointed out at this time. By defining a general purpose set of primitive routines to handle the data transfer, the actual means by which the bytes are transferred between slave and master is not visible to the calling routine. If the actual mechanism used needs to be altered the change will not affect the application software as long as the same external interface is maintained.

Another important feature of the primitive routines is the fact that they do not interpret the bytes that are sent to them. Due to this fact, the applications routines are free to send commands and parameters interspersed with the actual data. As an example, the terminal driver on an iSBC 544 board might perform format control based upon table information. The master applications software could use the data transfer primitives to transmit commands and parameters to the slave to update its format control information. Another advantage of the fact that the data is not interpreted is that it allows the calling routine to determine what data gets sent along. For instance, a specific terminal might be transmitting ASCII code while the master

Table 2
Byte Oriented Primitives

Primitive	Arguments	Usage
new$line	Queue$token Returns: ptr	Sets up a queue for byte oriented input. Ptr returned points to the first available byte.
next$space	Queue$token Returns: ptr	Increments the temporary give pointer to the next open space. Ptr returned either points to next byte or is zero specifying full queue.
back$space	Queue$token Returns: ptr	Decrements temporary give pointer. Ptr returned either points to byte or is unchanged indicating that the global give pointer was reached.
xmit	Queue$token Returns: status	Closes off a line entered via byte mode by updating global give ptr to equal temporary give ptr. Status is either "normal" or "null".
open$line	Queue$token Returns: ptr	Opens up a line for byte oriented output. Ptr returned either points to the next byte or is zero indicating an empty queue.
next$char	Queue$token Returns: ptr	Increments temporary take pointer. Ptr returned either points to next byte or is zero indicating an empty queue.
receive	Queue$token Returns: status	Closes off a line retrieved in byte mode by updating global take pointer to equal temporary ptr. Status is either "normal" or "null".

Table 3
Support Routines

Primitive	Arguments	Usage
get$status	Queue$token Returns: status	Returns status of queue. Possible values are "normal", "empty", "full" and "null".
set$interrupt	Queue$token, type Returns: status	Generates a slave → master or master → slave interrupt. Type code 0 is illegal and codes 8H → 0FH are reserved for use by the primitives.
set$handler	Queue$token, handler$adr Returns: status	Inserts address into vector table used for handling interrupts described above.
s$init	none	Called from slave software to initialize.
m$init	none	Called from master software to initialize.

software is expecting EBCDIC. The routine on the slave can very easily perform the necessary code conversion before stuffing the data into a queue.

Sample Slave Software

Given the existence of the primitive routines the applications routines on the slave and master can deal with the specific duties of each device. The following paragraphs revisit the code from application example 2, first for the slave and then for the master. Full code listings for these programs can be found in Appendix D.

The flowchart for the terminal input handler resident on the iSBC 544 board is shown in Figure 28. Support is provided for deleting characters (Rubout), deleting lines (control-X), pausing and resuming output (control-S and control-Q) and terminating lines (escape and carriage return). The sections of code reproduced below use this terminal input handler to present an example of the use of the data transfer primitives to enter and edit a line of input from a terminal. The byte variable *value* is based on the address variable *value$ptr* which is assigned by calls to the primitives. The routine *var$inp* inputs and returns a data byte from an I/O port specified by a calling parameter. This is necessary since the particular USART to be serviced is determined by reading the 8259A in-service register.

```
/* case 1; rubout; delete char */

do;
    new$ptr=back$space(token);
    if new$ptr=length$ptr then
        dummy=echo(token+1,.(bell),1);
    else
        do;
            dummy=echo(token+1,.(BS,SP,BS),3);
            ptr=new$ptr;
            count=count-1;
        end;
end;
```

Following this, the byte input is checked to see if it is a control character and if so a block within a DO CASE statement is executed. As an example of one of these blocks, if the character input was a RUBOUT the code sequence below is executed. The *back$space* primitive is called and a temporary pointer is returned to a location in the queue. A check is made to determine if the line was empty and, if so, a bell is echoed to signal the operator. If the pointer returned did not indicate an invalid RUBOUT the real pointer is assigned the value of the temporary pointer and a backspace, space, backspace is echoed to delete the previous character on the screen. Lastly, the character count for the current line is decremented.

```
VALUE$PTR=NEXT$SPACE(QUEUE$NUMBER);
VALUE=VAR$INP(USART$DATA$PORT(NUM));
```

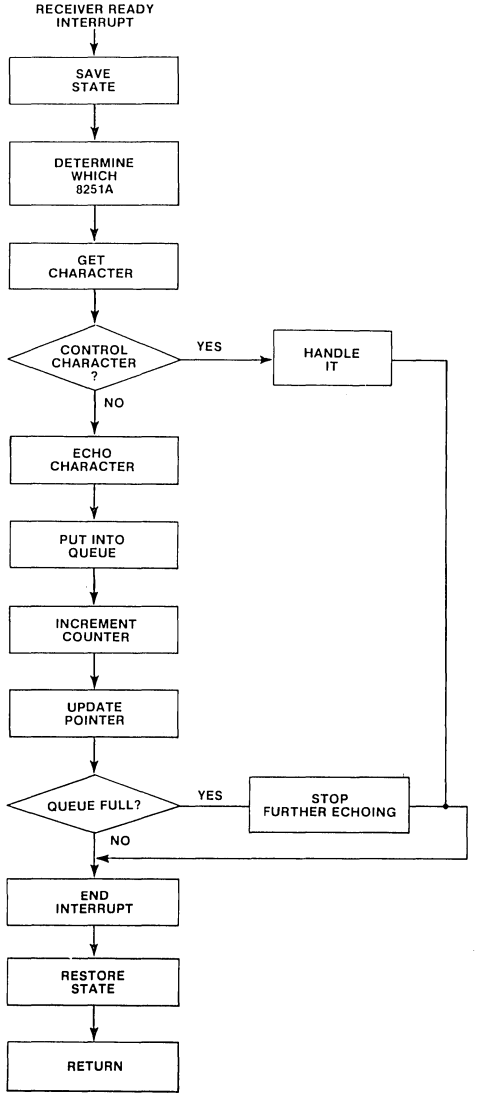


Figure 28. Flow Chart for Terminal Input Handler

In order to facilitate retrieval of the proper amount of information on the master side, the first byte of each message is defined to contain the number of characters in the message. Thus, when the master routine needs a line of input he uses the first byte as a count to retrieve the full line. The requirement for type-ahead is met by this mechanism since the number of lines in the queue at a given time is limited only by the length of the queue. When a full line of input is finished, the terminal input handler generates a slave to master interrupt to signal the master routine who may be waiting for this event.

The flowchart for a minimal terminal output handler is shown in Figure 29. Upon receipt of a

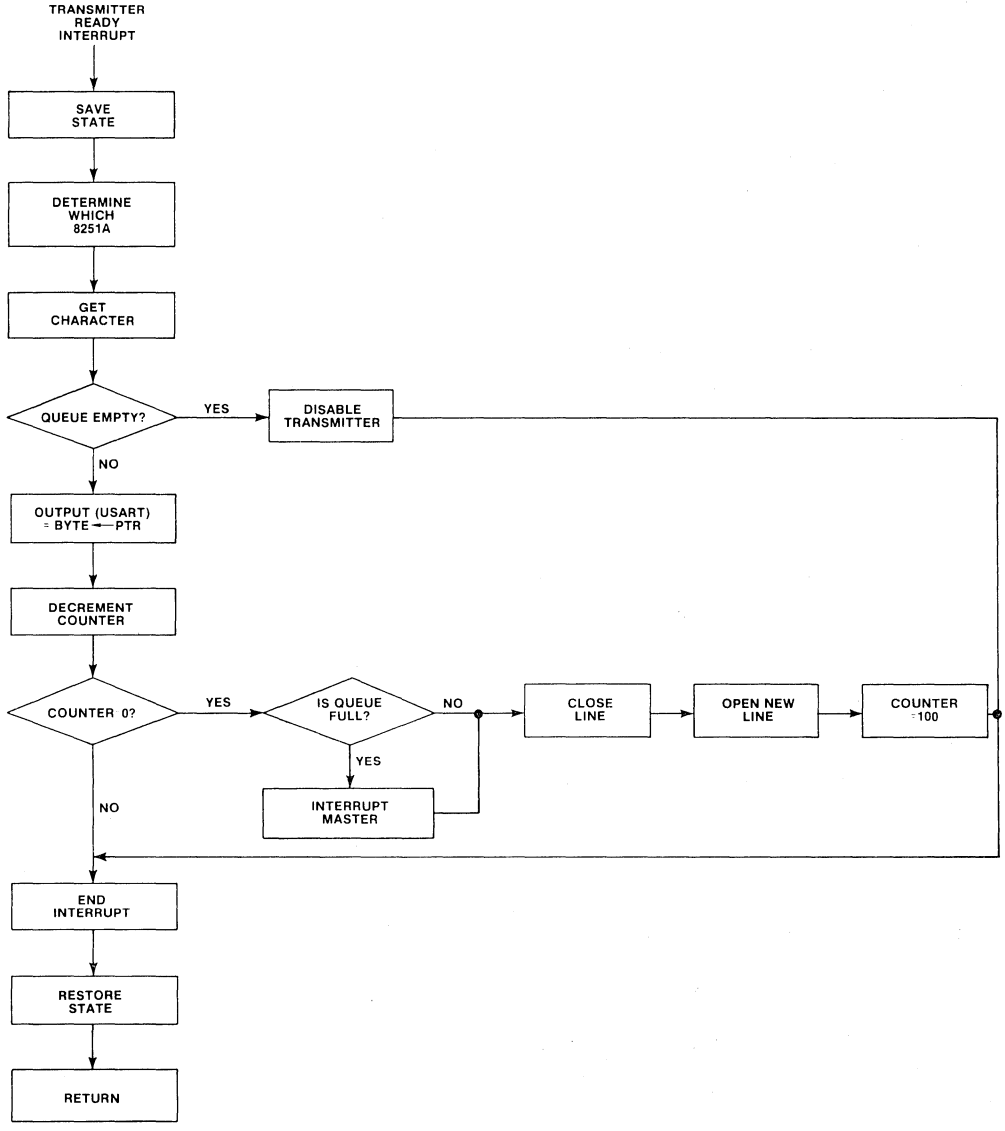


Figure 29. Flow Chart for Terminal Output Handler

transmitter ready interrupt the output handler requests a character from the appropriate queue. If one is available it is output to the USART. If the queue is empty, the transmitter is disabled. Whenever the master routine sends a line into the queue it will generate an interrupt to signal the slave handler and the transmitter will be reenabled. A line is opened via a call to *open$line* and it is kept open until 100 characters have been retrieved via calls to *next$byte*. At this time the line is closed by a call to *receive* making the space available to be reused. After this, a new call to *open$line* starts the process over again. If the call to *get$status* shows that the queue was full prior to the call to receive, an interrupt is sent to the master to reawaken any routine that may have been waiting for room in the queue to become available.

Sample Master Software

The RMX/80 handler for the master single board computer that will communicate with the software on the iSBC 544 board is diagrammed in Figure 30. In addition, the RMX/80 message used to convey information to the handler is shown on the right. The full software diagram is illustrated in Figure 31.

The input driver tasks execute a reentrant routine that services a request exchange that is specified in an initialization block that is unique to each of the input tasks. The necessary information is extracted from the request message and the *get$line* primitive is called upon to get a line of input from the queue. If the call to *get$line* for the length byte is unsuccessful the input task waits at

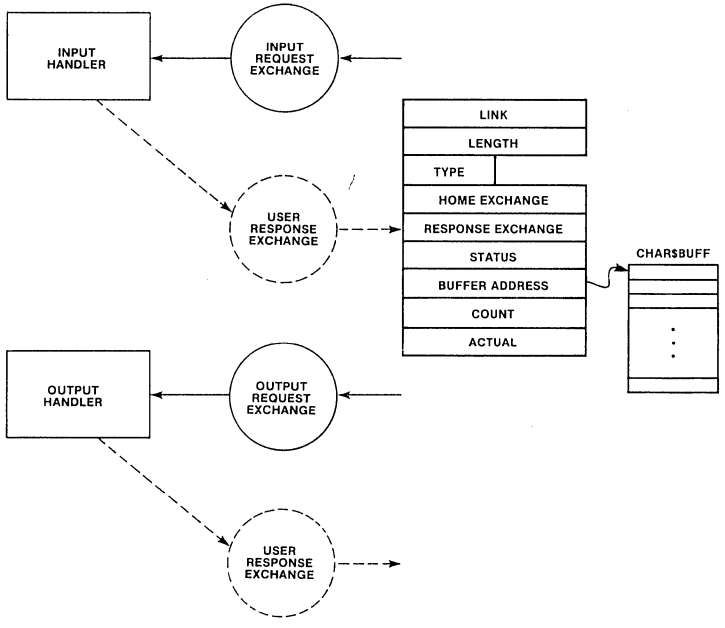


Figure 30. RMX/80 Handler for iSBC 544 Board

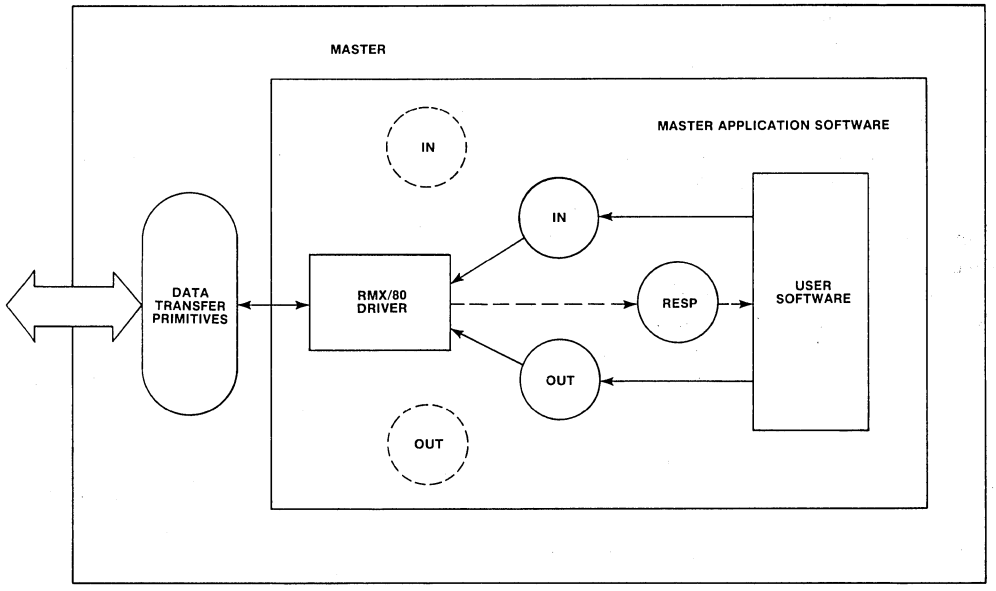


Figure 31. Master Software with RMX/80 Handler

the appropriate signal exchange for an interrupt from the slave indicating that a line is now available. Once the request is fulfilled the actual and status fields are set and the message is sent to the response exchange specified by the user.

The output handler performs in a manner very similar to the input handler. Upon receipt of a request message the handler attempts to transfer the characters from the user buffer to the appropriate queue. If the attempt is unsuccessful (ie. the queue has insufficient room available) the handler sends as many characters as will fit (count - overflow) and then waits for an interrupt from the slave indicating that room has been made available. This process is repeated until all of the data has been transmitted. As soon as the operation is complete the status field is cleared and the message is returned to the user specified response exchange.

Since the number of iSBC 544 slaves in the system is variable as are the memory base address, device programming information and queue sizes, some means of providing configuration information to the RMX/80 handlers is needed. This information resides in the *memory$allocation$module*. Public variables are declared in this module that are used by the RMX/80 tasks to determine how many devices (and therefore how many tasks need to be created) are in the system and where in the system address space their dual port RAM is located. In addition, queue sizes and device programming information are specified here.

VIII. SUMMARY

The intent of this application note has been to introduce the reader to the concept of the intelligent slave architecture and show the versatility of the first product based upon this architecture, the iSBC 544 Intelligent Communications Controller. The hardware and software aspects of the device were studied and results of benchmark tests were presented and studied. Finally, two example applications were worked out using the product as both a stand-alone controller and as an intelligent slave.

The bottom line is that the iSBC 544 controller, due to the advanced architecture around which it is designed, can be the means to the end for any application that requires communication. The dual nature of the controller provides the full power of a single board computer to the small application while the large system can make use of the fully programmabale intelligent slave to free the CPU for complicated processing duties.

I would like to extend my gratitude to Dave Jurasek for the work on the throughput testing and to Jack Tyler Inman for aid in the design of the system software.

APPENDIX A

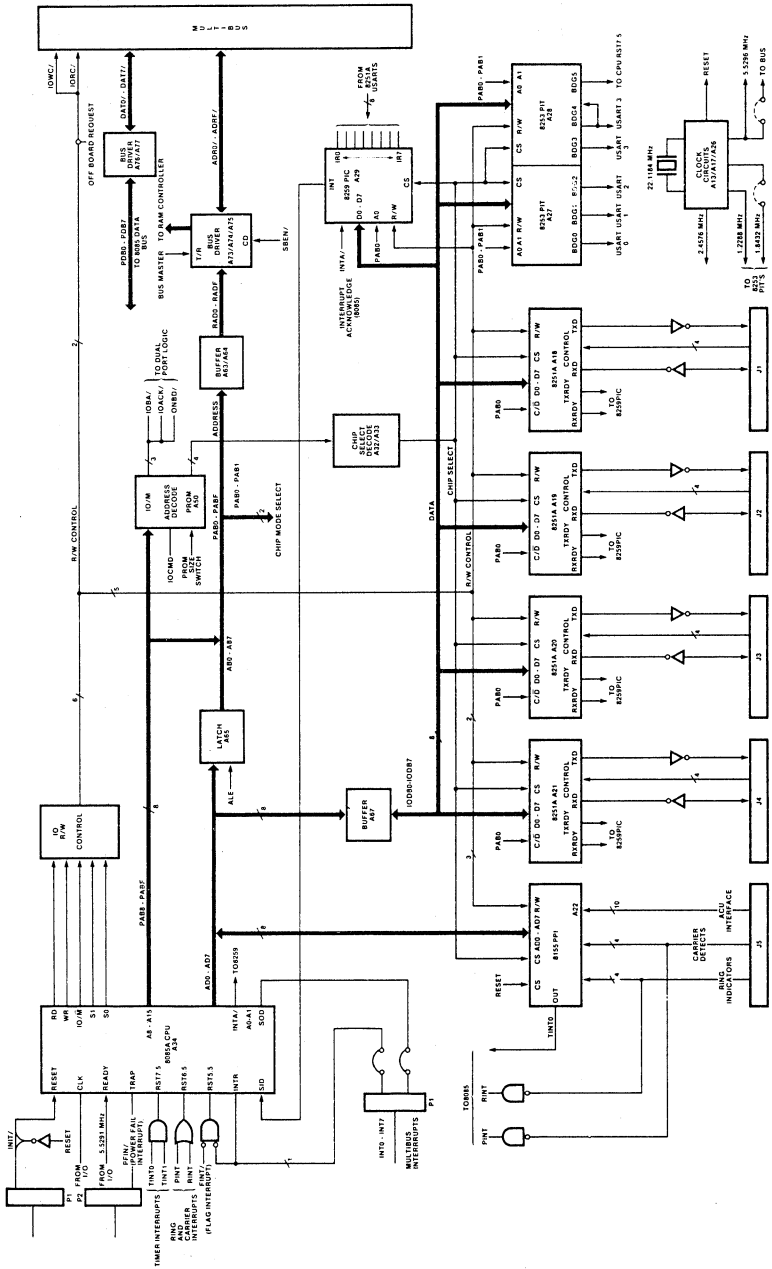


Figure A-1. iSBC 544 Input/Output and Interrupt Block Diagram

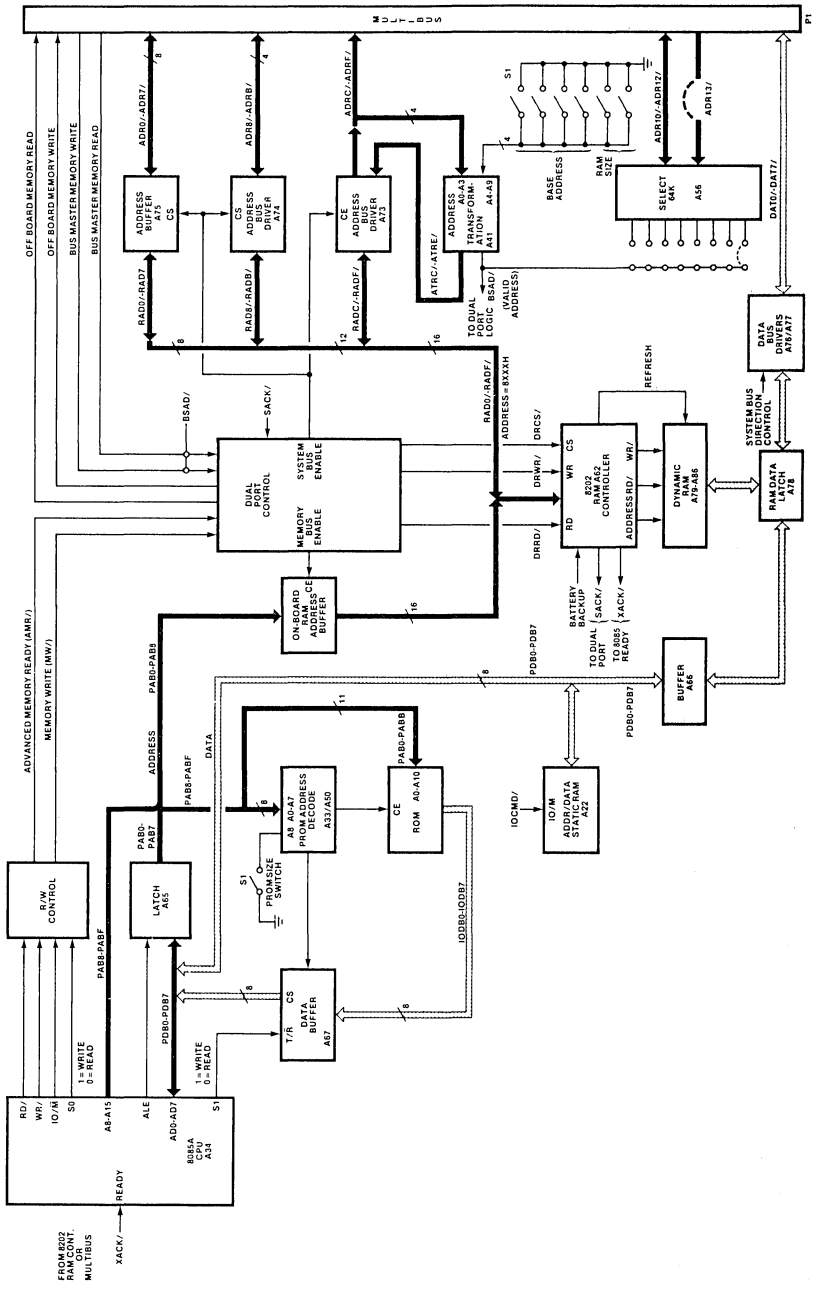


Figure A-2. iSBC 544 Memory Block Diagram

APPENDIX B

```
A M80 :F1:DMA544.M80


ISIS-II 8080/8085 MACRO ASSEMBLER, V3.0          MODULE   PAGE    1

  LOC  OBJ          LINE        SOURCE STATEMENT

                       1 $MOD85
  0080                 2 BASE    EQU     80H             ;BASE ADDRESS OF 204
  00E4                 3 MMSET   EQU     0E4H            ;MASTER MODE SET ADDRESS
  00E5                 4 MMRSET  EQU     0E5H            ;MASTER MODE RESET ADDRESS
  0052                 5 READ    EQU     52H             ;READ COMMAND CODE
  40FF                 6 TCOUNT  EQU     40FFH           ;TERMINAL COUNT AND DMA MODE OF
  0004                 7 DMAMOD  EQU     04H             ;DMA MODE WORD
  0001                 8 TADDR   EQU     1               ;TRACK ADDRESS
  0002                 9 SADDR   EQU     2               ;SECTOR ADDRESS
                      10         DSEG
  0000                11 BUFFER: DS      128             ;SECTOR BUFFER
                      12         ;
                      13         ;       TEST OF CAPABILITY FOR 544 TO SHARE MULTIBUS
                      14         ;       WITH OTHER MASTERS. ROUTINE PROGRAMS THE 204
                      15         ;       BOARD, INITIATES A READ TRANSFER, WAITS FOR
                      16         ;       AN INTERRUPT AND THEN TRAPS TO ICE85 BREAK-
                      17         ;       POINT AT 200.
                      18         ;
                      19         ASEG
  002C                20         ORG 2CH                 ;RST 5.5 ENTRY POINT
  002C D3E4           21         OUT     MMSET           ;SET MASTER MODE
  002E DB81           22         IN      BASE+1          ;GET RESULT
  0030 AF             23         XRA     A               ;SET FLAGS
  0031 C43900    C    24         CNZ     ERRTRP          ;NON-ZERO RESULT; ERROR TRAP
  0034 FB             25         EI                      ;REENABLE
  0035 C9             26         RET                     ;CONTINUE ON
                      27         ;
                      28         ;       MAINLINE ROUTINE
                      29         ;
                      30         EXTRN   INIT24          ;204 INITIALIZATION ROUTINE
                      31         EXTRN   WAITC           ;WAIT FOR 204 NOT BUSY ROUTINE
                      32         EXTRN   WAITP           ;WAIT FOR 204 PARAMETER REGIST
                      33         CSEG
  0000 F3             34         DI                      ;DISABLE
  0001 31FFBF         35         LXI     SP,0BFFFH       ;SET STACK POINTER
  0004 D3E4           36         OUT     MMSET           ;SET MASTER MODE FLIP FLOP
  0006 CD0000    E    37         CALL    INIT24          ;INITIALIZE 204
  0009 3E04           38         MVI     A,DMAMOD        ;SET DMA MODE
  000B D388           39         OUT     BASE+8          ;
  000D 3EFF           40         MVI     A,LOW(TCOUNT)   ;SET CONTROL REGISTER
  000F D385           41         OUT     BASE+5          ;
  0011 3E40           42         MVI     A,HIGH(TCOUNT)  ;
  0013 D385           43         OUT     BASE+5          ;
  0015 3E00      D    44         MVI     A,LOW(BUFFER)   ;OUTPUT LOW BYTE OF DMA ADDRESS
  0017 D384           45         OUT     BASE+4          ;
  0019 3E00      D    46         MVI     A,HIGH(BUFFER)  ;OUTPUT HIGH BYTE OF DMA ADDRESS
  001B D384           47         OUT     BASE+4          ;
  001D CD0000    E    48         CALL    WAITC           ;
  0020 3E52           49         MVI     A,READ          ;OUTPUT READ COMMAND
  0022 D380           50         OUT     BASE+0          ;
  0024 CD0000    E    51         CALL    WAITP           ;
  0027 3E01           52         MVI     A,TADDR         ;TRACK ADDRESS
  0029 D381           53         OUT     BASE+1          ;
  002B CD0000    E    54         CALL    WAITP           ;
  002E 3E02           55         MVI     A,SADDR         ;SECTOR ADDRESS
  0030 D381           56         OUT     BASE+1          ;
  0032 D3E5           57         OUT     MMRSET          ;RESET MASTER MODE FLIP/FLOP
```

```
0034 FB            58          EI                      ;ENABLE
0035 76            59          HLT                     ;AND HALT       ; WAIT FOR INTEF
0036 C30002        60          JMP     200H            ;TRAP TO ICE85 BREAKPOINT AT 200
                   61 ERRTRP:                          ;ERROR TRAP
0039 76            62          HLT                     ;FOR NOW
                   63          END


PUBLIC SYMBOLS


EXTERNAL SYMBOLS
INIT24 E 0000    WAITC  E 0000    WAITP  E 0000

USER SYMBOLS
BASE   A 0080    BUFFER D 0000    DMAMOD A 0004    ERRTRP C 0039    INIT24 E 0000    MF
READ   A 0052    SADDR  A 0002    TADDR  A 0001    TCOUNT A 40FF    WAITC  E 0000    WAI
```

```
A M80 :F1:INIT24.M80


ISIS-II 8080/8085 MACRO ASSEMBLER, V3.0          MODULE   PAGE    1

  LOC  OBJ          LINE        SOURCE STATEMENT

                       1 $MOD85
  0080                 2 BASE    EQU     80H             ;BASE ADDRESS OF 204
  00E4                 3 MMSET   EQU     0E4H            ;MASTER MODE SET ADDRESS
  00E5                 4 MMRSET  EQU     0E5H            ;MASTER MODE RESET ADDRESS
  0069                 5 SEEK    EQU     69H             ;SEEK COMMAND
  0035                 6 SPECFY  EQU     35H             ;"SPECIFY" COMMAND CODE
  0010                 7 BADTR1  EQU     10H             ;SPECIFY BAD TRACKS SURFACE 1
  0018                 8 BADTR2  EQU     18H             ;SPECIFY BAD TRACKS SURFACE 2
  00FF                 9 NOBAD   EQU     0FFH            ;NO BAD TRACKS
  00FF                10 CTADDR  EQU     0FFH            ;CURRENT TRACK ADDRESS NOT KNOWN
  000D                11 CHARS   EQU     0DH             ;SPECIFY DRIVE CHARACTERISTICS
  0008                12 SETTLE  EQU     08H             ;HEAD SETTLE TIME(SA800)
  0008                13 STEP    EQU     08H             ;STEP RATE
  0009                14 LOAD    EQU     09H             ;HEAD LOAD TIME
  4000                15 TCOUNT  EQU     4000H           ;TERMINAL COUNT AND DMA MODE  OF
  0004                16 DMAMOD  EQU     04H             ;DMA MODE WORD
  0080                17 BUSY    EQU     80H             ;204 BUSY MASK
  0020                18 PARFUL  EQU     20H             ;204 PARAMETER REGISTER FULL  AS
  0010                19 RESFUL  EQU     10H             ;204 RESULT BYTE FULL MASK
                      20         ;
                      21         ;       204 INITIALIZATION ROUTINE. RESETS 204 BOARD
                      22         ;       AND PERFORMS ALL OF THE NECESSARY INITIALIZATION
                      23         ;       OF THE 8257 AND 8271.
                      24         ;
                      25         CSEG
                      26         PUBLIC  INIT24          ;ENTRY POINT
                      27         PUBLIC  WAITC           ;WILL BE USED EXTERNALLY
                      28         PUBLIC  WAITP           ;
                      29 INIT24:
  0000 F3             30         DI                      ;DISABLE
  0001 3E0E           31         MVI     A,0EH           ;ENABLE 5.5 INTERRUPT
  0003 30             32         SIM                     ;
  0004 D3E4           33         OUT     MMSET           ;SET MASTER MODE FLIP FLOP
  0006 D38F           34         OUT     BASE+15         ;RESET INTERFACE
  0008 3E01           35         MVI     A,1             ;RESET 204
  000A D382           36         OUT     BASE+2          ;
  000C AF             37         XRA     A               ;
  000D D382           38         OUT     BASE+2          ;
  000F CD9900    C    39         CALL    WAITC           ;WAIT TILL COMMAND WRITE VALI
  0012 3E35           40         MVI     A,SPECFY        ;OUTPUT "SPECIFY" COMMAND
  0014 D380           41         OUT     BASE+0          ;
  0016 CDA100    C    42         CALL    WAITP           ;WAIT TILL PARAMETER WRITE VA IC
  0019 3E0D           43         MVI     A,CHARS         ;SPECIFYING DRIVE CHARACTERISTIC
  001B D381           44         OUT     BASE+1          ;
  001D CDA100    C    45         CALL    WAITP           ;
  0020 3E08           46         MVI     A,STEP          ;OUTPUT STEP RATE
  0022 D381           47         OUT     BASE+1          ;
  0024 CDA100    C    48         CALL    WAITP           ;
  0027 3E08           49         MVI     A,SETTLE        ;OUTPUT HEAD SETTLE TIME
  0029 D381           50         OUT     BASE+1          ;
  002B CDA100    C    51         CALL    WAITP           ;
  002E 3E09           52         MVI     A,LOAD          ;OUTPUT HEAD LOAD TIME
  0030 D381           53         OUT     BASE+1          ;
  0032 CD9900    C    54         CALL    WAITC           ;
```

APPENDIX B (Continued)

```
0035 3E35          55          MVI     A,SPECFY        ;SPECIFY BAD TRACKS
0037 D380          56          OUT     BASE+0          ;
0039 CDA100    C   57          CALL    WAITP           ;
003C 3E10          58          MVI     A,BADTR1        ;BAD TRACKS FOR SURFACE 1
003E D381          59          OUT     BASE+1          ;
0040 CDA100    C   60          CALL    WAITP           ;
0043 3EFF          61          MVI     A,NOBAD         ;FIRST TRACK
0045 D381          62          OUT     BASE+1          ;
0047 CDA100    C   63          CALL    WAITP           ;
004A 3EFF          64          MVI     A,NOBAD         ;SECOND BAD TRACK
004C D381          65          OUT     BASE+1          ;
004E CDA100    C   66          CALL    WAITP           ;
0051 3EFF          67          MVI     A,CTADDR        ;CURRENT TRACK ADDRESS (NOT F OW
0053 D381          68          OUT     BASE+1          ;
0055 CD9900    C   69          CALL    WAITC           ;
0058 3E35          70          MVI     A,SPECFY        ;
005A D380          71          OUT     BASE+0          ;
005C CDA100    C   72          CALL    WAITP           ;
005F 3E18          73          MVI     A,BADTR2        ;SURFACE 2
0061 D381          74          OUT     BASE+1          ;
0063 CDA100    C   75          CALL    WAITP           ;
0066 3EFF          76          MVI     A,NOBAD         ;FIRST TRACK
0068 D381          77          OUT     BASE+1          ;
006A CDA100    C   78          CALL    WAITP           ;
006D 3EFF          79          MVI     A,NOBAD         ;SECOND TRACK
006F D381          80          OUT     BASE+1          ;
0071 CDA100    C   81          CALL    WAITP           ;
0074 3EFF          82          MVI     A,CTADDR        ;CURRENT TRACK ADDRESS (NOT F OW
0076 D381          83          OUT     BASE+1          ;
0078 CD9900    C   84          CALL    WAITC           ;
007B 3E69          85          MVI     A,SEEK          ;SEEK TO TRACK 0
007D D380          86          OUT     BASE+0          ;
007F CDA100    C   87          CALL    WAITP           ;
0082 3E00          88          MVI     A,0             ;
0084 D381          89          OUT     BASE+1          ;
0086 D3E5          90          OUT     MMRSET          ;GO TO SLEEP WHILE 204 DOES IT
0088 FB            91          EI                      ;ENABLE INTERRUPTS
0089 76            92          HLT                     ;SLEEP
008A F3            93          DI                      ;DISABLE
008B 3E04          94          MVI     A,DMAMOD        ;SET DMA MODE
008D D388          95          OUT     BASE+8          ;
008F 3E00          96          MVI     A,LOW(TCOUNT)   ;SET CONTROL REGISTER
0091 D385          97          OUT     BASE+5          ;
0093 3E40          98          MVI     A,HIGH(TCOUNT)  ;
0095 D385          99          OUT     BASE+5          ;
0097 FB           100          EI                      ;
0098 C9           101          RET                     ;RETURN
                  102          ;
                  103          ;       WAITC AND WAITP ROUTINES
                  104          ;
0099 DB80         105 WAITC:   IN      BASE+0          ;GET STATUS BYTE
009B E680         106          ANI     BUSY            ;BUSY?
009D C29900    C  107          JNZ     WAITC           ;YES,LOOP
00A0 C9           108          RET                     ;NO,RETURN
                  109          ;
00A1 DB80         110 WAITP:   IN      BASE+0          ;GET STATUS REGISTER
00A3 E620         111          ANI     PARFUL          ;PARAMETER BUFFER FULL?
00A5 C2A100    C  112          JNZ     WAITP           ;YES,LOOP
00A8 C9           113          RET                     ;NO,RETURN
                  114          END


PUBLIC SYMBOLS
INIT24 C 0000    WAITC  C 0099    WAITP  C 00A1
```

APPENDIX B (Continued)

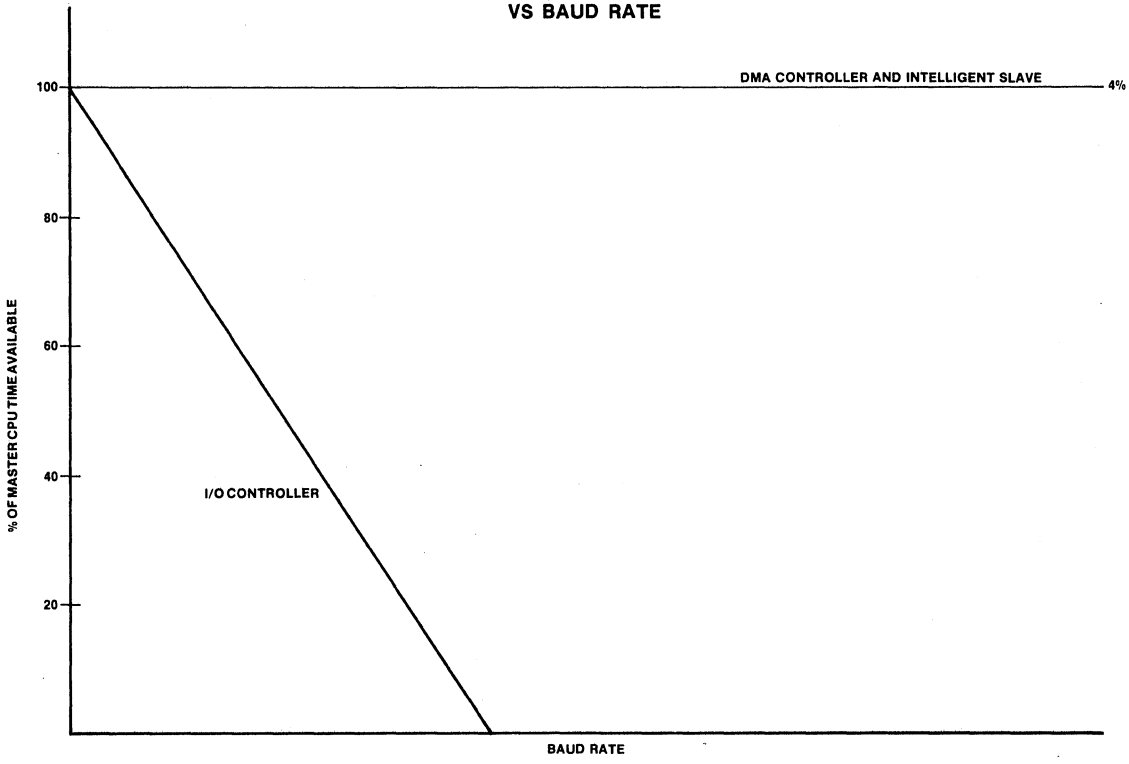
```
EXTERNAL SYMBOLS


USER SYMBOLS
BADTR1 A 0010    BADTR2 A 0018    BASE   A 0080    BUSY   A 0080    CHARS  A 000D    CTA
INIT24 C 0000    LOAD   A 0009    MMRSET A 00E5    MMSET  A 00E4    NOBAD  A 00FF     AI
SEEK   A 0069    SETTLE A 0008    SPECFY A 0035    STEP   A 0008    TCOUNT A 4000    WAI
```

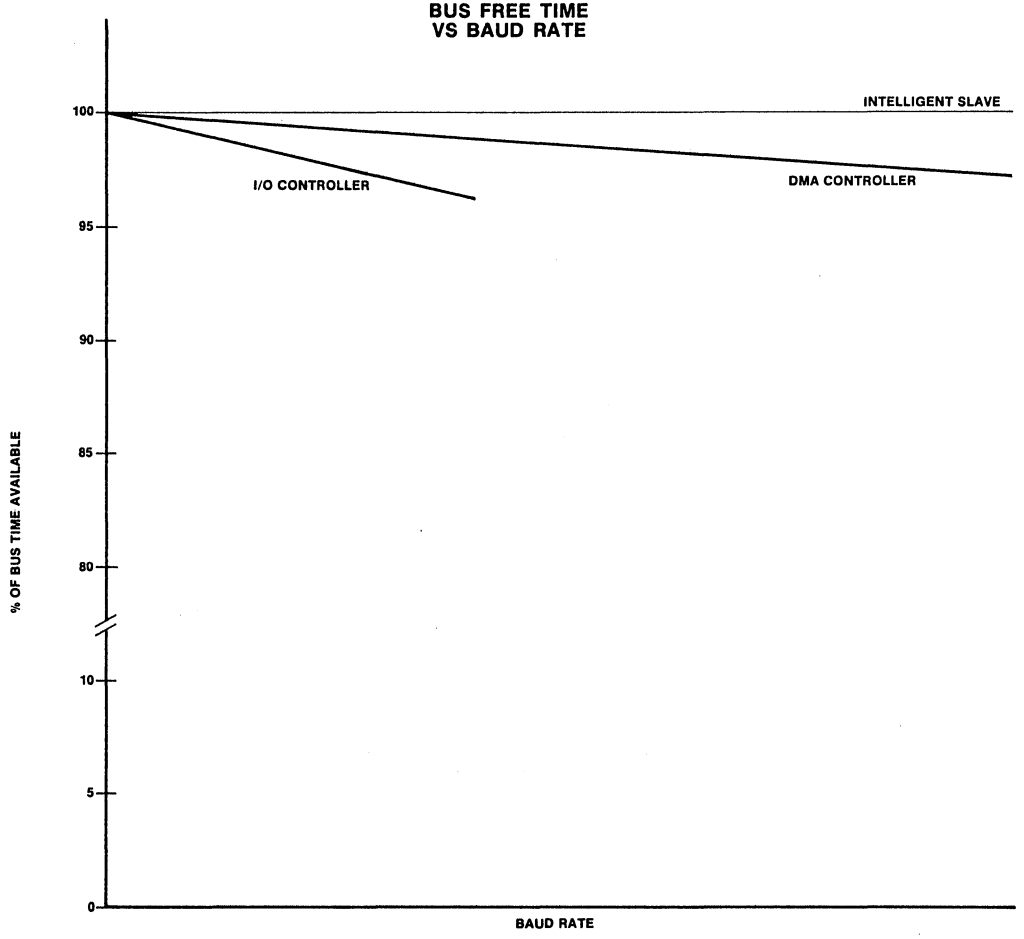
APPENDIX C

APPENDIX C (Continued)

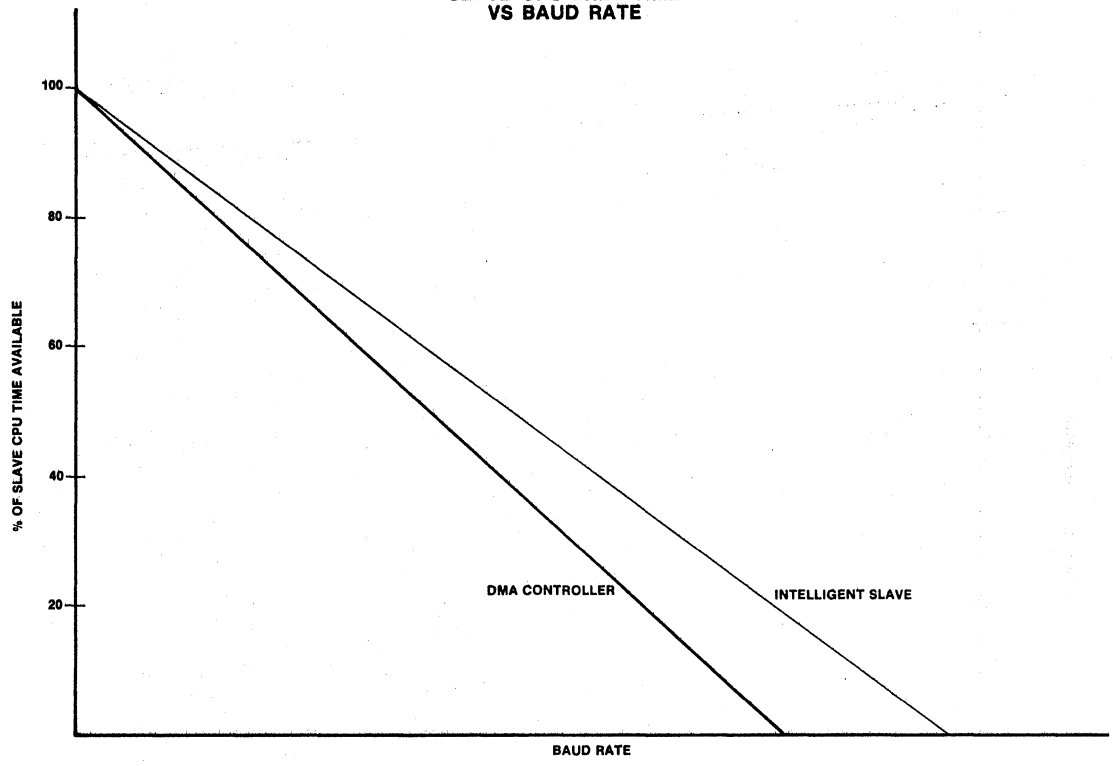
GRAPH 1
MASTER CPU FREE TIME
VS BAUD RATE



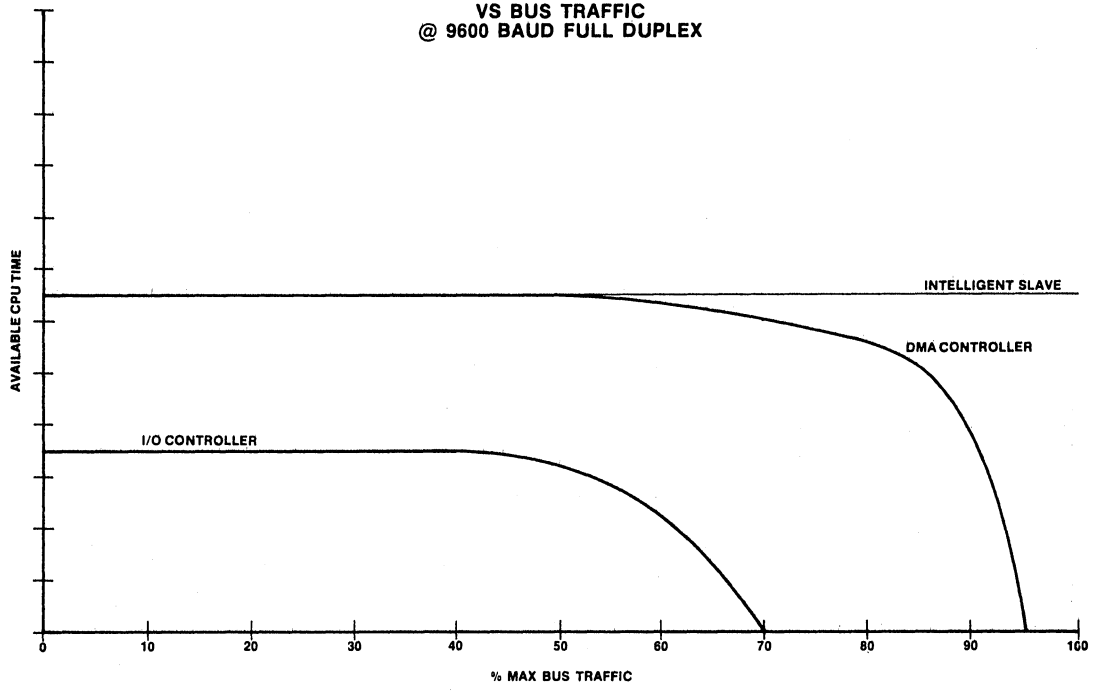
GRAPH 2
BUS FREE TIME
VS BAUD RATE



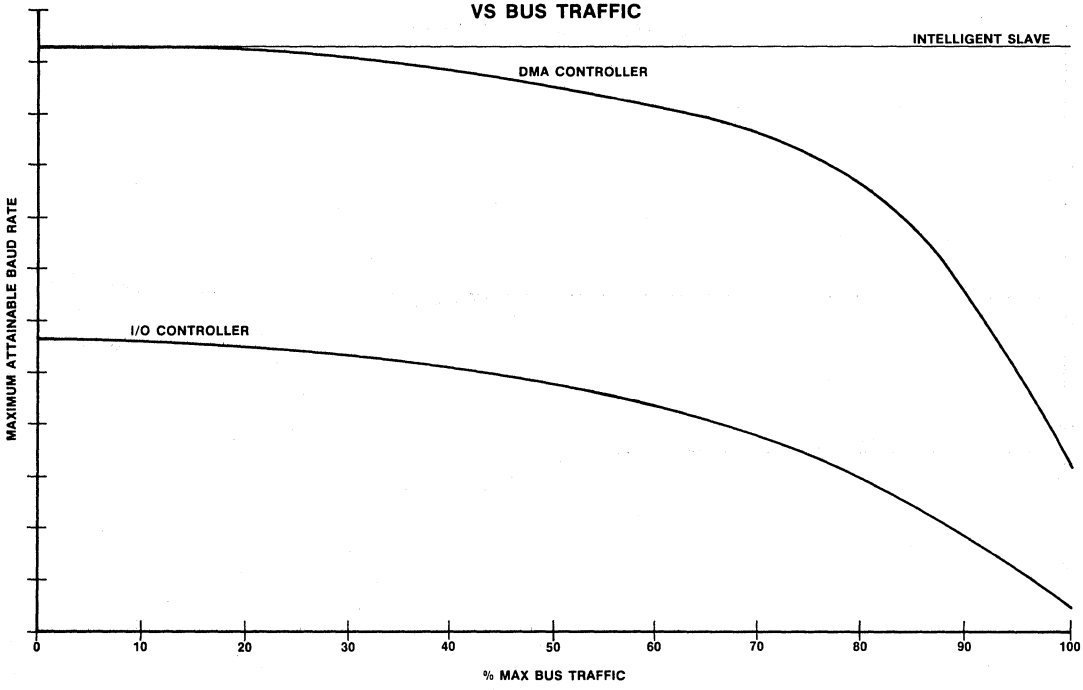
GRAPH 3
SLAVE CPU FREE TIME
VS BAUD RATE



GRAPH 4
COMMUNICATIONS PROCESSOR FREE TIME
VS BUS TRAFFIC
@ 9600 BAUD FULL DUPLEX



GRAPH 5
MAXIMUM BAUD RATE
VS BUS TRAFFIC



APPENDIX D

```
PL/M-80 COMPILER     SLAVE MAINLINE ROUTINE                          PAGE   1


ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE MAINLINE
OBJECT MODULE PLACED IN :F1:MAINLN.OBJ
COMPILER INVOKED BY:  PLM80 :F1:MAINLN.PLM PRINT(:F5:MAINLN.LST) PAGEWIDTH(78)


                $title('slave mainline routine')
    1           main$line:
                    DO;

                /*
                    Mainline routine. Sets up stack$ptr, calls s$init to init-
                    ialize queues,initializes some of the hardware, sets up the
                    initial flag interrupt handlers, and then halts with interru
              -   pts
                    enabled allowing the rest of the system to operate totally
                    in interrupt mode.
                */

                $nolist

   13   1       initial$handler:    PROCEDURE EXTERNAL;
   14   2       END initial$handler;


   15   1           DECLARE
                        command$word    LITERALLY   '41h',
                        port$a$8155 LITERALLY   'Ø0e9h',
                        command$8155    LITERALLY   'Ø0e8h',
                        mask$8259   LITERALLY   'Ø0e7h',
                        icw1$8259   LITERALLY   'Ø0e6h',
                        icw2$8259   LITERALLY   'Ø0e7h',
                        ocw3$8259   LITERALLY   'Ø0e6h',
                        read$isr    LITERALLY   'Ø0bh',
                        mask$word BYTE PUBLIC,
                        port$a$value BYTE PUBLIC,
                        stat    BYTE,
                        i BYTE;

   16   1           output(icw1$8259)=Ø0f6h;
   17   1           output(icw2$8259)=Ø0fh;
   18   1           output(mask$8259),mask$word=Ø0ffh;

   19   1           CALL s$init;

                /* set up 8259 for ISR reads */

   20   1           output(ocw3$8259)=read$isr;

   21   1           output(command$8155)=command$word;
   22   1           output(port$a$8155),port$a$value=Ø0c0h;
   23   1           DO i=Ø TO 7;
   24   2               stat=set$handler(i,.initial$handler);
```

```
  25   2          END;

  26   1          DO WHILE 1;
  27   2              HALT;
  28   2          END;

  29   1      END main$line;


MODULE INFORMATION:

     CODE AREA SIZE     = 004DH     77D
     VARIABLE AREA SIZE = 0004H      4D
     MAXIMUM STACK SIZE = 0002H      2D
     72 LINES READ
     0 PROGRAM ERROR(S)
```

```
P /M-80 COMPILER    SLAVE APPLICATION LEVEL SIGNAL HANDLE                PAGE   1


ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INITIALHANDLER
OBJECT MODULE PLACED IN :F1:FINTRT.OBJ
COMPILER INVOKED BY:  PLM80 :F1:FINTRT.PLM PRINT(:F5:FINTRT.LST) PAGEWIDTH(78)


                $title('slave application level signal handler')
    1           initial$handler:
                    DO;

                /*
                    Fields application level flag interrupts from the
                    master. If the type=go$type the device attached to the queue
                    specified is initialized with programming info sent into
                    the queue by the master. If the type is data$available the
                    specified transmitter is enabled unless a control$s pause
                    is in effect.
                */

                $nolist

   32   1           DECLARE
                        no$pause    LITERALLY   '1',
                        go$type     LITERALLY   '1',
                        data$available  LITERALLY   '2',
                        enable$xmit LITERALLY   '1',
                        reset       LITERALLY   '40h',
                        timer$1$command$port LITERALLY  '0dbh',
                        timer$2$command$port LITERALLY  '0dfh',
                        mask$8259   LITERALLY   '0e7h',
                        mask$word BYTE EXTERNAL,
                        mask (8) BYTE DATA(
                            0fch,
                            0fch,
                            0f3h,
                            0f3h,
                            0cfh,
                            0cfh,
                            03fh,
                            03fh),
                        transmitter$state (8) BYTE PUBLIC,
                        type    BYTE,
                        token   BYTE,
                        i   BYTE,
                        prog$info (5) BYTE,
                        actual ADDRESS,
                        usart$command$port (8) BYTE EXTERNAL,
                        usart$state (8) BYTE PUBLIC,
                        length$pointer (8) ADDRESS PUBLIC,
                        pointer (8) ADDRESS PUBLIC,
                        char$count (8) BYTE PUBLIC,
                        timer$load$port (8) BYTE DATA(
                            0d8h,
```

```
                        0d8h,
                        0d9h,
                        0d9h,
                        0dah,
                        0dah,
                        0dch,
                        0dch);

33   1      initial$handler:    PROCEDURE (code) PUBLIC;

34   2          DECLARE code BYTE;

35   2          token=code AND 0fh;
36   2          type=shr(code,4);

37   2          IF type=go$type THEN
38   2          DO;
39   3              transmitter$state(token)=no$pause;
           /* reset usart */

40   3              DO i=0 TO 3;
41   4                  CALL varout(usart$command$port(token),0);
42   4              END;

43   3              CALL varout(usart$command$port(token),reset);

44   3              actual=get$line(token,.prog$info,5);

           /* program the devices */

45   3              CALL varout(usart$command$port(token),prog$info(0));
46   3              CALL varout(usart$command$port(token),usart$state(token)
       -    :=prog$info(1));
47   3              IF token < 7 THEN
48   3                  CALL varout(timer$1$command$port,prog$info(2));
                    ELSE
49   3                  CALL varout(timer$2$command$port,prog$info(2));
50   3              CALL varout(timer$load$port(token),prog$info(3));
51   3              CALL varout(timer$load$port(token),prog$info(4));

           /* open up the four input queues for data input */

52   3              length$pointer(token-1)=new$line(token-1);
53   3              pointer(token-1)=next$space(token-1);
54   3              char$count(token-1)=0;
55   3              output(mask$8259),mask$word=mask$word AND mask(token);
56   3          END;

               ELSE
57   2          IF (type=data$available) AND (transmitter$state(token)=no$pa
       -    use) THEN
58   2          DO;
59   3              usart$state(token)=usart$state(token) OR enable$xmit;
60   3              CALL varout(usart$command$port(token),usart$state(token)
       -    );
61   3          END;
```

```
                RETURN;
  63    2         END;

  64    1     END initial$handler;

MODULE INFORMATION:

     CODE AREA SIZE     = 0182H    386D
     VARIABLE AREA SIZE = 0043H     67D
     MAXIMUM STACK SIZE = 0004H      4D
     154 LINES READ
     0 PROGRAM ERROR(S)
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INPUTHANDLER
OBJECT MODULE PLACED IN :F1:INHDLR.OBJ
COMPILER INVOKED BY:  PLM80 :F1:INHDLR.PLM PRINT(:F5:INHDLR.LST) PAGEWIDTH(78)


                $nointvector title('slave terminal input handler')
  1             input$handler:
                    DO;

                /*
                    544 resident interrupt service routine. After receiver
                    ready interrupt the 8259 In Service Register(ISR) is
                    read to determine which device is requesting service.
                    The character is read in and placed in the appropriate
                    queue. A check is made for break characters and appropriate
                    action is taken if any are found. When an endline character
                    is encountered the length byte is filled in ( it was left
                    vacant when the line was started) and the xmit primitive is
                    called to update the global queue pointer to permit access
                    to the line. At this time the master is signalled to signify
                    that a new line is available for processing.
                */

                $nolist

  34    1           DECLARE
                        control$x   LITERALLY   '18H',
                        control$s   LITERALLY   '13H',
                        control$q   LITERALLY   '11H',
                        rubout      LITERALLY   '7FH',
                        escape      LITERALLY   '1BH',
                        CR      LITERALLY   '0DH',
                        LF      LITERALLY   '0AH',
                        BS      LITERALLY   '08H',
                        SP      LITERALLY   '20H',
                        bell        LITERALLY   '07H',
                        ptr     LITERALLY   'pointer(token)',
                        length$ptr  LITERALLY   'length$pointer(token)',
                        count       LITERALLY   'char$count(token)',
                        disable$xmit    LITERALLY   '0FEH',
                        enable$xmit LITERALLY   '01H',
                        no$pause        LITERALLY   '1',
                        pause   LITERALLY   '0',
                        line$available  LITERALLY   '1',
                        ocw2$8259   LITERALLY   '0E6H',
                        ocw3$8259   LITERALLY   '0E6H',
                        EOI     LITERALLY   '20H';

  35    1           DECLARE
                        value$ptr   ADDRESS,
                        value BASED value$ptr BYTE,
                        line$length BASED value$ptr BYTE,
                        dummy   ADDRESS,
                        ISR BYTE,
                        token   BYTE,
```

```
                    stat    BYTE,
                    new$ptr ADDRESS;

36   1          DECLARE
                    pointer (8) ADDRESS EXTERNAL,
                    length$pointer (8) ADDRESS EXTERNAL,
                    char$count (8) BYTE EXTERNAL,
                    usart$state (8) BYTE EXTERNAL,
                    usart$command$port (8) BYTE EXTERNAL,
                    usart$data$port (8) BYTE EXTERNAL,
                    transmitter$state (8) BYTE EXTERNAL;

37   1      index:  PROCEDURE (value) BYTE;
38   2          DECLARE value BYTE;

39   2          IF value=control$x THEN RETURN 0;
41   2          IF value=rubout THEN RETURN 1;
43   2          IF value=control$s THEN RETURN 2;
45   2          IF value=control$q THEN RETURN 3;
47   2          IF value=escape THEN RETURN 4;
49   2          IF value=CR THEN RETURN 5;
51   2          RETURN 6;
52   2      END;

53   1      echo:   PROCEDURE(token,buf$ptr,num$char) ADDRESS;

54   2          DECLARE (buf$ptr,num$char,actual) ADDRESS,
                    token BYTE;

55   2          actual=send$line(token,buf$ptr,num$char);
56   2          usart$state(token)=usart$state(token) OR enable$xmit;
57   2          CALL varout(usart$command$port(token),usart$state(token));
58   2          RETURN actual;
59   2          END;

60   1      delete$line:    PROCEDURE;

61   2          length$ptr=new$line(token);
62   2          ptr=next$space(token);
63   2          count=0;
64   2          dummy=echo(token+1,.('#',CR,LF),3);
65   2          RETURN;
66   2      END;

67   1      end$line:   PROCEDURE;

68   2          value$ptr=length$ptr;
69   2          line$length=count;
70   2          ptr=next$space(token);
71   2          stat=xmit(token);
72   2          length$ptr=new$line(token);
73   2          ptr=next$space(token);
74   2          count=0;
75   2          stat=set$s$interrupt(token,line$available);
76   2          RETURN;
77   2      END;
```

```
 78   1      in$hdlr:    PROCEDURE INTERRUPT 0 PUBLIC;
 79   2          ISR=input(ocw3$8259);

 80   2          token=6;

 81   2      again:
              ISR=shl(ISR,2);

 82   2          IF NOT carry THEN
 83   2              DO;
 84   3              IF token=0 THEN RETURN; /* no bits set */
                     ELSE
 86   3                  DO;
 87   4                      token=token-2;
 88   4                      GOTO again;
 89   4                  END;

 90   3              END;
 91   2          value$ptr=ptr;
 92   2          value=varinp(usart$data$port(token)) AND 07fh;

 93   2          DO CASE index(value);

                     /* case 0; control$x; delete line */

 94   3              DO;
 95   4                  CALL delete$line;
 96   4              END;

                     /* case 1; rubout; delete char */

 97   3              DO;
 98   4                  new$ptr=back$space(token);
 99   4                  IF new$ptr=length$ptr THEN
100   4                      dummy=echo(token+1,.(bell),1);
                         ELSE
101   4                      DO;
102   5                          dummy=echo(token+1,.(BS,SP,BS),3);
103   5                          ptr=new$ptr;
104   5                          count=count-1;
105   5                      END;
106   4              END;

                     /* case 2; control$s; pause output */

107   3              DO;
108   4                  usart$state(token+1)=usart$state(token+1) AND disabl
          -   e$xmit;
109   4                  CALL varout(usart$command$port(token+1),usart$state(
          -   token+1));
110   4                  transmitter$state(token+1)=pause;
111   4              END;

                     /* case 3; control$q; resume output */
112   3              DO;
113   4                  usart$state(token+1)=usart$state(token+1) OR enable$
```

```
          -    xmit;
 114    4                  CALL varout(usart$command$port(token+1),usart$state(
          -    token+1));
 115    4                  transmitter$state(token+1)=no$pause;
 116    4              END;

                       /* case 4; escape; terminate line */
 117    3              DO;
 118    4                  dummy=echo(token+1,.('#',CR,LF),3);
 119    4                  value=CR;
 120    4                  count=count+1;
 121    4                  CALL end$line;
 122    4              END;

                       /* case 5; carriage return; terminate line */
 123    3              DO;
 124    4                  dummy=echo(token+1,.(CR,LF),2);
 125    4                  count=count+1;
 126    4                  ptr=next$space(token);
 127    4                  value$ptr=ptr;
 128    4                  value=LF;
 129    4                  count=count+1;
 130    4                  CALL end$line;
 131    4              END;

                       /* case 6; non-break character; stuff into queue */
 132    3              DO;
 133    4                  dummy=echo(token+1,ptr,1);
 134    4                  ptr=next$space(token);
 135    4                  IF ptr=0 THEN CALL delete$line; /* full buffer */
 137    4                  ELSE count=count+1;
 138    4              END;

 139    3          END; /* of do case */
 140    2          output(ocw3$8259)=EOI;

 141    2          RETURN;

 142    2          END;

 143    1      END input$handler;


MODULE INFORMATION:

     CODE AREA SIZE     = 0398H    920D
     VARIABLE AREA SIZE = 0011H     17D
     MAXIMUM STACK SIZE = 0010H     16D
     255 LINES READ
     0 PROGRAM ERROR(S)
```

```
P /M-80 COMPILER    SLAVE CHARACTER OUTPUT HANDLER                    PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE OUTPUTHANDLER
OBJECT MODULE PLACED IN :F1:OUTHLR.OBJ
COMPILER INVOKED BY:  PLM80 :F1:OUTHLR.PLM PRINT(:F5:OUTHLR.LST) PAGEWIDTH(78)


                $nointvector title('slave character output handler')
   1            output$handler:
                    DO;

                /*
                    544 resident interrupt service routine. After transmitter
                    ready interrupt, 8259 In Service Register(ISR) is read to
                    determine which device is requesting service. A character
                    is requested from the appropriate queue and, if available,
                    is sent to the usart. If the queue is empty the transmitter
                    is disabled pending a signal from the master when more
                    characters are put into the queue.
                */

                $nolist

  11    1           DECLARE
                        ocw2$8259   LITERALLY   '0E6H',
                        ocw3$8259   LITERALLY   '0E6H',
                        disable$xmit    LITERALLY   '0FEH',
                        true        LITERALLY   '0FFH',
                        false       LITERALLY   '00H',
                        EOI     LITERALLY   '0A0H';

  12    1           DECLARE
                        ISR BYTE,
                        token   BYTE,
                        actual  ADDRESS,
                        value   BYTE;

  13    1           DECLARE
                        usart$state (8) BYTE EXTERNAL,
                        usart$command$port (8) BYTE PUBLIC DATA(
                            0D1H,
                            0D1H,
                            0D3H,
                            0D3H,
                            0D5H,
                            0D5H,
                            0D7H,
                            0D7H),
                        usart$data$port (8) BYTE PUBLIC DATA(
                            0D0H,
                            0D0H,
                            0D2H,
                            0D2H,
                            0D4H,
```

```
                        0D4H,
                        0D6H,
                        0D6H);

 14   1      out$hlr:    PROCEDURE INTERRUPT 1 PUBLIC;

             /* get active level number and use it to determine queue$token *
          -   /

 15   2          ISR=input(ocw3$8259);

 16   2          token=7;

 17   2      again:
             ISR=shl(ISR,1);
 18   2          IF NOT carry THEN
 19   2              DO;
 20   3                  IF token=1 THEN RETURN; /* no bits in ISR set */
                         ELSE
 22   3                      DO;
 23   4                          token=token-2;
 24   4                          ISR=shl(ISR,1);
 25   4                          GOTO again;
 26   4                      END;
 27   3              END;

 28   2              actual=get$line(token,.value,1);
 29   2              IF actual=0 THEN
 30   2                  DO; /* empty queue. Disable transmitter */
 31   3                      usart$state(token)=usart$state(token) AND disabl
          -   e$xmit;
 32   3                      CALL varout(usart$command$port(token),usart$stat
          -   e(token));
 33   3                  END;
                     ELSE
 34   2                  CALL varout(usart$data$port(token),value);
 35   2              output(ocw3$8259)=EOI;
 36   2              RETURN;
 37   2              END;
 38   1      END output$handler;
```

```
MODULE INFORMATION:

     CODE AREA SIZE     = 00A4H    164D
     VARIABLE AREA SIZE = 0005H      5D
     MAXIMUM STACK SIZE = 000CH     12D
     102 LINES READ
     0 PROGRAM ERROR(S)
```

```
PL/M-80 COMPILER    RMX/80-544 INITIALIZATION TASK                    PAGE   1


ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INIT544
OBJECT MODULE PLACED IN :F1:INIT54.OBJ
COMPILER INVOKED BY:  PLM80 :F1:INIT54.PLM PRINT(:F5:INIT54.LST) PAGEWIDTH(78)


                $title('rmx/80-544 initialization task')
   1            init$544:
                    DO;

                /*
                    Task code for 544 driver initialization task. Info
                    from application supplied memory allocation block
                    is accessed to set up queues and transfer device programming
                    info to the slave board(s) and create the required
                    service handler tasks.
                */

                $nolist

  56    1       input$driver:   PROCEDURE EXTERNAL;
  57    2       END input$driver;

  58    1       output$driver:  PROCEDURE EXTERNAL;
  59    2       END output$driver;

  60    1       signal: PROCEDURE EXTERNAL;
  61    2       END signal;

  62    1           DECLARE
                        stack$size  LITERALLY   '256',
                        go$type     LITERALLY   '1';

  63    1           DECLARE
                        ptr     ADDRESS,
                        init$table BASED ptr STRUCTURE(
                            base$adr ADDRESS,
                            queue$token BYTE,
                            prog$info (5) BYTE),
                        i       BYTE,
                        overflow    ADDRESS,
                        queue$init$table (1) STRUCTURE(
                            base$adr ADDRESS,
                            queue$size (8) ADDRESS) EXTERNAL,
                        initialization$table (1) BYTE EXTERNAL,
                        stat        BYTE,
                        num$devices     BYTE EXTERNAL,
                        num$boards  BYTE EXTERNAL,
                        service$exchange$table (1) ADDRESS EXTERNAL,
                        signal$exchange$table (1) ADDRESS EXTERNAL,
                        service$exchanges (1) BYTE EXTERNAL,
                        signal$exchanges (1) BYTE EXTERNAL,
                        task$descriptors (1) BYTE EXTERNAL,
```

```
                    stacks (1) BYTE EXTERNAL,
                    info$block (1) STRUCTURE(
                        base$adr ADDRESS,
                        queue$token BYTE,
                        index BYTE) EXTERNAL,
                    rqactv      ADDRESS EXTERNAL;

64   1          DECLARE
                    rom$input$std static$task$descriptor DATA(
                        'input ',
                        .input$driver,
                        0, /* stack will be assigned individually */
                        stack$size,
                        200,
                        0, /* tba */
                        0), /* tba */
                    rom$output$std static$task$descriptor DATA(
                        'output',
                        .output$driver,
                        0,
                        stacksize,
                        201,
                        0,
                        0),
                    input$hdlr$std static$task$descriptor,
                    output$hdlr$std static$task$descriptor;

65   1      init$xch:   PROCEDURE (xch$ptr);
            /* initializes expanded interrupt exchanges */

66   2          DECLARE xch$ptr ADDRESS,
                    xch BASED xch$ptr int$exchange$descriptor;

67   2          xch.link=.xch.link;
68   2          xch.type=int$type;
69   2          xch.length=5;
70   2          RETURN;
71   2          END;

72   1      init$54: PROCEDURE PUBLIC;

73   2          DO i=0 TO num$boards-1;
74   3              CALL m$init(.queue$init$table(i));
75   3          END;

76   2          CALL move(size(rom$input$std),.rom$input$std,.input$hdlr$std
         -   );
77   2          CALL move(size(rom$output$std),.rom$output$std,.output$hdlr$
         -   std);
78   2          ptr=.initialization$table;

79   2          DO i=0 TO num$devices*2 BY 2;
            /* send pogramming info to slave */
80   3              overflow=send$line(init$table.base$adr,init$table.queue$
         -   token,.init$table.prog$info,5);
81   3              stat=set$m$interrupt(init$table.base$adr,init$table.queu
```

```
          -    e$token,go$type);

               /* create service and signal exchanges */

   82    3              CALL rqcxch(service$exchange$table(i):=.service$exchange
          -    s+10*i);
   83    3              CALL rqcxch(service$exchange$table(i+1):=.service$exchan
          -    ges+10*(i+1));
   84    3              CALL init$xch(.signal$exchanges+15*i);
   85    3              CALL init$xch(.signal$exchanges+15*(i+1));
   86    3              CALL rqcxch(signal$exchange$table(i):=.signal$exchanges+
          -    15*i);
   87    3              CALL rqcxch(signal$exchange$table(i+1):=.signal$exchange
          -    s+15*(i+1));

   88    3              info$block(i).base$adr,
                        info$block(i+1).base$adr=init$table.base$adr;
   89    3              info$block(i).queue$token=init$table.queue$token-1;
   90    3              info$block(i+1).queue$token=init$table.queue$token;
   91    3              info$block(i).index=i;
   92    3              info$block(i+1).index=i+1;

   93    3              input$hdlr$std.sp=.stacks+stack$size*i;
   94    3              output$hdlr$std.sp=.stacks+stack$size*(i+1);
   95    3              input$hdlr$std.exchange$address=.info$block(i);
   96    3              output$hdlr$std.exchange$address=.info$block(i+1);
   97    3              input$hdlr$std.task$ptr=.task$descriptors+20*i;
   98    3              output$hdlr$std.task$ptr=.task$descriptors+20*(i+1);

   99    3              CALL rqctsk(.input$hdlr$std);
  100    3              CALL rqctsk(.output$hdlr$std);
  101    3              ptr=ptr+8;
  102    3           END;

  103    2           CALL rqsetv(.signal,2);
  104    2           CALL rqelvl(2);
  105    2           CALL rqsusp(rqactv);
  106    2        END; /* of task */

  107    1        END init$544;


MODULE INFORMATION:

     CODE AREA SIZE     = 02C3H    707D
     VARIABLE AREA SIZE = 002AH     42D
     MAXIMUM STACK SIZE = 0006H      6D
     285 LINES READ
     0 PROGRAM ERROR(S)
```

```
P /M-80 COMPILER    RMX/80-544 INITIALIZATION MODULE AND                PAGE   1


ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INITMODULE
OBJECT MODULE PLACED IN :F1:MAB.OBJ
COMPILER INVOKED BY:  PLM80 :F1:MAB.PLM PRINT(:F5:MAB.LST) PAGEWIDTH(78)


            $title('rmx/80-544 initialization module and memory allocation b
          -  lock')
   1        init$module:
                DO;

            /*
                Initialization tables created and allocation of memory for 5
          -  44
                handler done here.
            */

   2    1       DECLARE
                    number$of$devices   LITERALLY   '4',
                    baud$rate$count$l   LITERALLY   '32',
                    baud$rate$count$h   LITERALLY   '00',
                    usart$mode      LITERALLY   '4eh',
                    usart$cmd       LITERALLY   '27h',
                    ctr$0$mode      LITERALLY   '36h',
                    ctr$1$mode      LITERALLY   '76h',
                    ctr$2$mode      LITERALLY   '0b6h',
                    ctr$3$mode      LITERALLY   '36h',
                    num$devices BYTE PUBLIC DATA(number$of$devices-1),
                    num$boards BYTE PUBLIC DATA(1),
                    service$exchange$table (8) ADDRESS PUBLIC,
                    signal$exchange$table (8) ADDRESS PUBLIC,
                    signal$type (8) BYTE PUBLIC,
                    service$exchanges (80) BYTE PUBLIC,
                    signal$exchanges (120) BYTE PUBLIC,
                    task$descriptors (160) BYTE PUBLIC,
                    stacks (2048) BYTE PUBLIC,
                    info$block (32) BYTE PUBLIC,
                    queue$init$table (1) STRUCTURE(
                        base$adr ADDRESS,
                        queue$size (8) ADDRESS) PUBLIC DATA(
                            0e000h,
                            256,
                            1765,
                            256,
                            1765,
                            256,
                            1765,
                            256,
                            1765),
                    base$table (1) ADDRESS PUBLIC DATA(
                            0e000h),
                    initialization$table (number$of$devices) STRUCTURE(
                        base$adr    ADDRESS,
```

```
                        queue$token BYTE,
                        prog$info (5) BYTE) PUBLIC DATA(
                            0e000h,
                            1,
                            usart$mode,
                            usart$cmd,
                            ctr$0$mode,
                            baud$rate$count$l,
                            baud$rate$count$h,

                            0e000h,
                            3,
                            usart$mode,
                            usart$cmd,
                            ctr$1$mode,
                            baud$rate$count$l,
                            baud$rate$count$h,

                            0e000h,
                            5,
                            usart$mode,
                            usart$cmd,
                            ctr$2$mode,
                            baud$rate$count$l,
                            baud$rate$count$h,

                            0e000h,
                            7,
                            usart$mode,
                            usart$cmd,
                            ctr$3$mode,
                            baud$rate$count$l,
                            baud$rate$count$h);
  3   1       END init$module;


MODULE INFORMATION:

     CODE AREA SIZE     = 0036H     54D
     VARIABLE AREA SIZE = 09B0H   2480D
     MAXIMUM STACK SIZE = 0000H      0D
     79 LINES READ
     0 PROGRAM ERROR(S)
```

```
P /M-80 COMPILER    SLAVE->MASTER INTERRUPT HANDLER                    PAGE   1


ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE SIGNALHANDLER
OBJECT MODULE PLACED IN :F1:SIGNAL.OBJ
COMPILER INVOKED BY:  PLM80 :F1:SIGNAL.PLM PRINT(:F5:SIGNAL.LST) PAGEWIDTH(78)


                $nointvector title('slave->master interrupt handler')
   1            signal$handler:
                    DO;

                /*
                    Fields all slave->master signals(interrupts) and calls rqisn
              -  d
                    with the proper signal exchange address.
                */

                $nolist

  26    1           DECLARE
                        i       BYTE,
                        ptr     ADDRESS,
                        (flag BASED ptr) BYTE,
                        num$boards  BYTE EXTERNAL,
                        num$devices BYTE EXTERNAL,
                        signal$type (1) BYTE EXTERNAL,
                        index       BYTE,
                        token       BYTE,
                        signal$exchange$table (1) ADDRESS EXTERNAL,
                        base$table (1) ADDRESS EXTERNAL;

  27    1       signal: PROCEDURE INTERRUPT 2 PUBLIC;

                /* poll slave boards and find one generating interrupt */

  28    2           i=0;

  29    2       next:
                    ptr=base$table(i)+1;
  30    2           IF flag=0 THEN
  31    2               DO;
  32    3                   i=i+1;
  33    3                   IF i > num$boards THEN RETURN; /* erroneous signal *
              -  /
  35    3                   ELSE GOTO next;
  36    3               END;

                /* get queue token and use it to index into signal exchange tabl
              -  e */

  37    2           token=(flag AND 0fh);
  38    2           index=4*i+token;

                /* if index is out of range don't attempt the isend */
```

```
 39   2          IF index <= num$devices THEN
 40   2          DO;
 41   3              CALL rqisnd(signal$exchange$table(index));
 42   3              signal$type(index)=shr(flag,4);
 43   3          END;

                 ELSE
 44   2              CALL rqendi;

             /* zero flag to acknowledge interrupt */

 45   2          flag=0;
 46   2          RETURN;
 47   2          END;

 48   1      END signal$handler;
```

MODULE INFORMATION:

```
     CODE AREA SIZE     = 008BH    139D
     VARIABLE AREA SIZE = 0005H      5D
     MAXIMUM STACK SIZE = 000AH     10D
     110 LINES READ
     0 PROGRAM ERROR(S)
```

```
P /M-80 COMPILER    RMX/80-544 INPUT SERVICE HANDLER                    PAGE   1


ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INPUTDRIVER
OBJECT MODULE PLACED IN :F1:INPUT.OBJ
COMPILER INVOKED BY:  PLM80 :F1:INPUT.PLM PRINT(:F5:INPUT.LST) PAGEWIDTH(78)


                $title('rmx/80-544 input service handler')
    1           input$driver:
                    DO;

                /*
                    Master resident task code. Monitors service exchange
                    and fills input requests by retrieving characters from
                    the proper queue(board$base and device info is passed
                    via default exchange field). By definition the first byte
                    of a line of input contains the length of that line.
                    This figure is used to retrieve the exact number of characte
            -    rs
                    available in a given line.
                */

                $nolist

   27   1           DECLARE
                        rqactv      ADDRESS EXTERNAL,
                        td BASED rqactv task$descriptor,
                        service$exchange$table (1) ADDRESS EXTERNAL,
                        signal$exchange$table (1) ADDRESS EXTERNAL;

   28   1       input$driver:   PROCEDURE REENTRANT PUBLIC;

   29   2           DECLARE
                        service$exchange    ADDRESS,
                        board$base  ADDRESS,
                        queue$token BYTE,
                        signal$exchange ADDRESS,
                        msg$ptr     ADDRESS,
                        msg BASED msg$ptr th$msg,
                        actual      ADDRESS,
                        dummy       ADDRESS,
                        info$block$ptr  ADDRESS,
                        info$block BASED info$block$ptr STRUCTURE(
                            base$adr    ADDRESS,
                            queue$token BYTE,
                            index       BYTE),
                        num$char    BYTE,
                        stat        BYTE;

                /* get info out of default field */

   30   2           info$block$ptr=td.exchange$address; /* default exchange fiel
            -    d */
   31   2           service$exchange=service$exchange$table(info$block.index);
```

```
 32   2          board$base=info$block.base$adr;
 33   2          queue$token=info$block.queue$token;
 34   2          signal$exchange=signal$exchange$table(info$block.index);
 35   2          DO forever;

             /* wait for request message */

 36   3              msg$ptr=rqwait(service$exchange,0);

 37   3      retry:
             /* try to get line count out of queue */
                     actual=get$line(board$base,queue$token,.num$char,1);

             /* if unsuccessful wait for signal and try again */

 38   3              IF actual=0 THEN
 39   3              DO;
 40   4                  dummy=rqwait(signal$exchange,0);
 41   4                  GOTO retry;
 42   4              END;

             /* if all okay get line */

 43   3              actual=get$line(board$base,queue$token,msg.buffer$adr,nu
          -   m$char);
 44   3              msg.actual=actual;
 45   3              msg.status=0;
 46   3              CALL rqsend(msg.resp$ex,msg$ptr);
 47   3          END; /* of do forever */

 48   2      END; /* of task */

 49   1      END input$driver;
```

```
MODULE INFORMATION:

     CODE AREA SIZE      = 012CH    300D
     VARIABLE AREA SIZE  = 0000H      0D
     MAXIMUM STACK SIZE  = 0017H     23D
     171 LINES READ
     0 PROGRAM ERROR(S)
```

```
P /M-80 COMPILER    RMX/80-544 OUTPUT SERVICE HANDLER                PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE OUTPUTDRIVER
OBJECT MODULE PLACED IN :F1:OUTPUT.OBJ
COMPILER INVOKED BY:  PLM80 :F1:OUTPUT.PLM PRINT(:F5:OUTPUT.LST) PAGEWIDTH(78)


                $title('rmx/80-544 output service handler')
   1            output$driver:
                    DO;

                /*
                    Master resident task code. Monitors service exchange and
                    fills output requests by stuffing characters into the approp
              -   riate
                    queue. If insufficient room is available the task waits
                    for 1 second and retries up to 100 times after which it
                    signals a time out error. If the transmission completes
                    successfully the slave is signalled to indicate that data is
              -
                    available.
                */

                $nolist

  39    1           DECLARE
                        data$available  LITERALLY   '2',
                        time$out    LITERALLY   '1';

  40    1           DECLARE
                        rqactv  ADDRESS EXTERNAL,
                        (td BASED rqactv) task$descriptor,
                        service$exchange$table (1) ADDRESS EXTERNAL,
                        signal$exchange$table (1) ADDRESS EXTERNAL;

  41    1       output$driver:  PROCEDURE REENTRANT PUBLIC;

  42    2           DECLARE
                        service$exchange    ADDRESS,
                        signal$exchange     ADDRESS,
                        base$adr        ADDRESS,
                        queue$token     BYTE,
                        msg$ptr         ADDRESS,
                        msg BASED msg$ptr   th$msg,
                        tries$left      BYTE,
                        overflow        ADDRESS,
                        dummy           ADDRESS,
                        stat            BYTE,
                        info$block$ptr      ADDRESS,
                        info$block BASED info$block$ptr STRUCTURE(
                            base$adr    ADDRESS,
                            queue$token BYTE,
                            index       BYTE);
```

```
                /* initialize */

  43    2           info$block$ptr=td.exchange$address;
  44    2           service$exchange=service$exchange$table(info$block.index);
  45    2           signal$exchange=signal$exchange$table(info$block.index);
  46    2           base$adr=info$block.base$adr;
  47    2           queue$token=info$block.queue$token;

  48    2           DO forever;

                /* wait for request message */

  49    3                 msg$ptr=rqwait(service$exchange,0);
  50    3                 tries$left=100;
  51    3           retry:
                          overflow=send$line(base$adr,queue$token,msg.buffer$adr,m
          -    sg.count);
  52    3                 IF overflow <> 0 THEN
  53    3                 DO;
  54    4                     dummy=rqwait(signal$exchange,20);
  55    4                     tries$left=tries$left-1;
  56    4                     IF tries$left > 0 THEN GOTO retry;
                              ELSE
  58    4                         DO;
  59    5                             msg.status=time$out;
  60    5                             msg.actual=0;
  61    5                             GOTO quit;
  62    5                         END;
  63    4                 END;
  64    3                 msg.status=0;
  65    3                 stat=set$m$interrupt(base$adr,queue$token,data$available
          -    );
  66    3                 msg.actual=msg.count;
  67    3           quit:
                          CALL rqsend(msg.resp$ex,msg$ptr);
  68    3               END; /* of do forever */

  69    2           END; /* of task */

  70    1           END output$driver;


MODULE INFORMATION:

     CODE AREA SIZE     = 0159H    345D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0019H     25D
     198 LINES READ
     0 PROGRAM ERROR(S)
```

APPENDIX D (Continued)

```
A M80 :F1:CFG544.M80 PRINT(:F4:CFG544.LST) PAGEWIDTH(78) MACROFILE


ISIS-II 8080/8085 MACRO ASSEMBLER, V3.0         CFG544   PAGE    1

  LOC  OBJ          LINE        SOURCE STATEMENT

                       1          NAME    CFG544
                       2          CSEG
                       3          PUBLIC  RQRATE
  0000 0800            4 RQRATE: DW      8
                       5 $NOLIST
                     127 $LIST
                     128 $NOGEN
  0000               129 NTASK   SET     0
  0000               130 NEXCH   SET     0
                     131 ;
                     132 ;       BUILD THE INITIAL TASK TABLE
                     133 ;
                     134 ; --------\  THIS TASK IS NECESSARY FOR THE 544 HANDLE
                         R
                     135 ; --------/  IT CREATES EVERYTHING ELSE IT NEEDS.
                     136         STD     INIT54,200,1,0
                     191         STD     LINECH,64,130,0
                     246
                     247 ;       ALLOCATE TASK DESCRIPTORS
                     248 ;
                     249         GENTD
                     253 ;
                     254 ;       BUILD INITIAL EXCHANGE TABLE
                     255 ;
                     256         XCHADR  RESPEX
                     264 ;
                     265 ;       BUILD CREATE TABLE
                     266 ;
                     267         CRTAB
                     274         END


PUBLIC SYMBOLS
RQCRTB C 0026    RQRATE C 0000

EXTERNAL SYMBOLS
INIT54 E 0000    LINECH E 0000    RESPEX E 0000

USER SYMBOLS
CRTAB  + 0000    GENTD  + 0000    IET    C 0024    INIT54 E 0000
INTXCH + 0000    ITT    C 0002    LINECH E 0000    NEXCH  A 0001
NTASK  A 0002    PUBXCH + 0007    RESPEX E 0000    RQCRTB C 0026
RQRATE C 0000    STD    + 0000    TDBASE D 0108    XCH    + 0005
XCHADR + 0002
```